

S A L T · L A K E · C I T Y

1984
SUMMER CONFERENCE
PROCEEDINGS

USENIX

USENIX Association
Software Tools Users Group

Summer Conference

Salt Lake City 1984

PROCEEDINGS

June 12-15, 1984

Salt Lake City, Utah, USA

For additional copies of these proceedings, write:

USENIX Association

P.O. Box 7

El Cerrito, CA 94530 USA

Price \$25.00 plus \$5.00 for overseas airmail

© 1984 by The USENIX Association and Software Tools User Group

All rights reserved.

This volume is published as a collective work.

Rights to individual papers remain
with the author or author's employer.

UNIX is a trademark of Bell Laboratories.

Other trademarks noted in the text.

ACKNOWLEDGMENTS

Sponsored by:

USENIX Association
P.O. Box 7
El Cerrito, CA 94350 USA

In cooperation with:

Software Tools Users Group
1259 El Camino Real, #242
Menlo Park, CA 94025 USA

Hosted by:

Department of Computer Science
University of Utah

Conference Chair:

Randy Frank University of Utah

Program Co-Chairs:

Jay Lepreau University of Utah
Spencer Thomas University of Utah

Program Committee:

Steve Bellovin AT&T Bell Laboratories
Kirk McKusick U.C. Berkeley
Mike Muuss BRL
Dennis Ritchie AT&T Bell Laboratories

Software Tools Liaison:

Barbara Chase Hughes Aircraft

Tutorial Coordinator:

Bruce Borden Silicon Graphics

USENIX Conference Coordinator: Wally Wedel NBI

USENIX Meeting Planner: Judy DesHarnais USENIX Association

Proceedings Preparation:

Randy Frank University of Utah
Jay Lepreau University of Utah
Spencer Thomas University of Utah
Grant Weiler University of Utah

Table of Contents

Track A – Wednesday, June 13

UNIX Directions

Wed-1A (9:00 – 10:30 am)

General Chair: Randy Frank, Univ. of Utah

Program Chairs: Jay Lepreau and Spencer Thomas, Univ. of Utah

Opening Remarks

Conference Organizers and USENIX board

KEYNOTE ADDRESS: An Architecture History of the UNIX System xi
Stuart I. Feldman, Bell Communications Research
(Text of paper printed separately for insertion at this point.)

Towards a UNIX Standard 1
Michael Tilson, Human Computing Resources Corporation

Mail and News

Wed-2A (11:00 – 12:30 pm)

Chair: Steve Bellovin, AT&T Bell Labs

ACSNET – The Australian Alternative to UUCP 11
Piers Dick-Lauder and R.J. Kummerfeld (University of Sydney), Robert Elz (University of Melbourne)

Broadcasting of Netnews and Network Mail via Satellite 18
Lauren Weinstein

The Berkeley Internet Name Domain Server 23
Douglas B. Terry, Mark Painter, David W. Riggall, and Songnian Zhou, U.C. Berkeley

MMDF II: A Technical Review 32
Douglas P. Kingston III, Ballistic Research Laboratory

DRAGONMAIL: A Prototype Conversation-Based Mail System 42
Douglas E. Comer and Larry L. Peterson, Purdue University

Networks and Distributed Processing

Wed-3A (2:00 – 3:30 pm)

Chair: Jay Lepreau, Univ. of Utah

Converting the BBN TCP/IP to 4.2BSD 52
Robert Walsh and Robert Gurwitz, BBN Laboratories

Network Tasking in the LOCUS Distributed UNIX System 62
David Butterfield and Gerald Popek, Locus Computing Corporation

Project Athena 72
James Gettys, Digital Equipment Corporation – MIT/Project Athena

TEMPO – A Network Time Controller for a Distributed Berkeley UNIX System 78
Riccardo Gusella and Stefano Zatti, U.C. Berkeley

Distributed File Systems

Wed-4A (4:00 – 5:30 pm)	Chair: Mike Muuss, BRL
The Version 8 Network File System	86
<i>Peter J. Weinberger, AT&T Bell Laboratories</i>	
Towards a Distributed File System	87
<i>Walter F. Tichy and Zuwang Ruan, Purdue University</i>	
The Livermore Interactive Network Communication System	98
<i>Alex Phillips, Lawrence Livermore National Labs</i>	
Panel on Distributed File Systems	
<i>Session speakers and others</i>	

Track A – Thursday, June 14

Programming Languages

Thu-1A (9:00 – 10:30 am)	Chair: Kirk McKusick, U.C. Berkeley
An Optimizing Portable C Compiler for the New CDC CYBER 180	100
<i>Kok-Weng Lee and Mario D. Ruggiero, Human Computing Resources Corporation</i>	
A Simple Simulation Toolkit in "C"	110
<i>Robert P. Warnock III (Fortune Systems Corporation) and Bakul Shah</i>	
An Adaptable Object Code Optimizer for UNIX Systems	111
<i>Bill Appelbe and Bob Querido, U.C. San Diego & NCR Corporation</i>	
Using Modula-2 for System Programming with UNIX	119
<i>Michael L. Powell, Digital Equipment Corporation</i>	
The FP-Shell	133
<i>Manton Matthews and Yogeesh Kamath, University of South Carolina</i>	

Programming Environments and Window Systems

Thu-2A (11:00 – 12:30 pm)	Chair: Spencer Thomas, Univ. of Utah
SYSTANT: An Integrated Programming Environment for Modular C under UNIX	141
<i>Stowe Boyd, AZREX, Inc.</i>	
LIPs: Knowledge Base Development System	151
<i>Kiyoki Ohkubo, PANAFACOM Limited</i>	
WINDX – Windows for the UNIX Environment	159
<i>Peter E. Collins, Ithaca Intersystems, Inc.</i>	
The Maryland Window System	166
<i>Chris Torek and Mark Weiser, University of Maryland</i>	
A Text-Oriented Terminal Multiplexor for Blits	173
<i>Rob Pike, AT&T Bell Laboratories</i>	

Kernel I

Thu-3A (2:00 – 3:30 pm)

Chair: Dennis Ritchie, AT&T Bell Labs

A Multiprocessor UNIX System	174
<i>Maurice J. Bach and Steven J. Buroff, AT&T Bell Laboratories</i>	
A Demand-paging Virtual Memory Manager for System V	178
<i>Richard Miller, Human Computing Resources Corporation</i>	
Resource Controls, Privileges, and other MUSH	183
<i>Robert Elz, University Of Melbourne</i>	
A Dynamic Bad-Block Forwarding Algorithm	192
<i>Bakul Shah and Robert P. Warnock III (Fortune Systems Corporation)</i>	
An Expandable Object-Based UNIX Kernel	193
<i>Erik Reeh Nielsen, Søren Lauesen, and Vilhelm Rosenqvist, NCR Systems Engineering Copenhagen</i>	

Kernel II

Thu-4A (4:00 – 5:30 pm)

Chair: Lou Salkind, NYU

Processes as Files	203
<i>Thomas J. Killian, AT&T Bell Laboratories</i>	
Memory Management Units and the UNIX Kernel	208
<i>Clara S. Lai and Chris Peer Johnson, UniSoft Systems</i>	
Techniques for Debugging XENIX Device Drivers	214
<i>Paresh K. Vaish and Jean Marie McNamara, Intel Corporation</i>	
User-Mode Development of Hardware and Kernel Software	224
<i>Robert P. Warnock III, Fortune Systems Corporation</i>	

Track A – Friday, June 15

Performance Analysis and Comparisons

Fri-1A (9:00 – 10:30 am)

Chair: Clem Cole, Masscomp

Relating Benchmarks to Performance Projections, or What Do You Do With 20 Pounds of Benchmark Data?	227
<i>Gene Dronek, Aim Technology</i>	
UNIX System V and 4BSD Performance	228
<i>Jeffrey P. Lankford, AT&T Bell Laboratories</i>	
Measuring and Improving the Performance of 4.2BSD	237
<i>Sam Leffler (Lucasfilm, Ltd), Mike Karels, and M. Kirk McKusick (U.C. Berkeley)</i>	

Applications, Text Processing, Graphics

Fri-2A (11:00 – 12:30 pm)

Chair: Andrew Tannenbaum, Masscomp

The UNIX System HELP Facility <i>Thomas W. Butler and Lisa A. Kennedy, AT&T Bell Laboratories</i>	253
Adventures with Typesetter (Device) Independent Troff <i>Mark Kahrs and Lee Moore, University of Rochester</i>	258
The Readers Workbench – A System for Computer Assisted Reading <i>Evan L. Ivie, Brigham Young University</i>	270
Circuit Design Aids – CDA: A Printed Circuit Board Manufacturing System <i>Terry Slattery and Willie McCool, U.S. Naval Academy</i>	280
A Transition Diagram Editor <i>Charles C. Mills (U.C. Berkeley) and Anthony I. Wasserman (U.C. San Francisco)</i>	287

System Management and Porting

Fri-3A (2:00 – 3:30 pm)

Chair: Mike O'Dell, Group L Corp.

Optical Storage Management Under the UNIX Operating System <i>Perry S. Kivolowitz, SUNY Stony Brook</i>	297
Automatic Software Distribution <i>Andrew Koenig, AT&T Bell Laboratories</i>	312
4bsd UNIX TCP/IP and VMS DECNET: Experience in Negotiating a Peaceful Coexistence <i>Van Jacobson, Craig Leres, Joseph Sventek, and Wayne Graves, Lawrence Berkeley Laboratory</i>	323
Experiences with a Large Mixed-Language System Running Under the UNIX Operating System <i>Richard A. Becker, AT&T Bell Laboratories</i>	326
The Dynamics of a Semi-Large Software Project with Specific Reference to a UNIX System Port <i>Brian Pawlowski and Alan Filipinski, Motorola, Inc.</i>	332

Open Session

Fri-4A (4:00 – 5:00 pm)

Short unreviewed presentations.

Track B: Workshops, Software Tools, Projects, and Panels

Workshop: How to Teach UNIX

Wed-3B (2:00 – 3:30 pm)

Chair: Bubbette Mcleod, Informatics General Corp.

Workshop Abstract	343
<i>Panelists are Ms McLeod, Jay Hosler (User Training Corp.), Stan Kelly-Bootle (author of "The Devil's DP Dictionary"), Bob Nystrom (Momentum Compter Systems), and Jim Joyce (International Technical Seminars)</i>	
Introducing People to UNIX	344
<i>Bubbette Mcleod, Informatics General Corporation</i>	
Interactivity in Packaged UNIX Training: A Modest Proposal	346
<i>Jay Hosler, User Training Corporation</i>	

Workshop: TCP/IP and Networking

Thu-1B (9:00 – 10:30 am)

Chair: Mike Muuss, Ballistic Research Lab

Workshop Abstract	350
-----------------------------	-----

Software Tools Technical Session I

Thu-3B (2:00 – 3:30 pm)

Chair: Dave Martin, Hughes Aircraft Company

Avionics Simulation Package: A large System Application in Ratfor	352
<i>Dave Martin, Hughes Aircraft Company</i>	
An Update on the Software Tools Standards Effort	352
<i>Bill Meine, Sun Microsystems</i>	
Ada? Yet Another VOS?	353
<i>Neil Groundwater, Analytic Disciplines, Inc.</i>	
A LEX Tool for the VOS	353
<i>Vern Paxson, Real Time Systems Group, Lawrence Berkeley Labs</i>	

Software Tools Technical Session II

Thu-4B (4:00 – 5:30 pm)

Chair: Dave Martin, Hughes Aircraft Company

A Portable TOPS-20-like Command Parser	354
<i>Nelson Beebe, Univ. of Utah</i>	
Mine Planning Applications in Ratfor	354
<i>Mike Norred, MINESoft</i>	
Future Directions for STUG (open discussion)	355
<i>Dave Martin, moderator, Hughes Aircraft Company</i>	
A Ratfor Implementation of KERMIT	356
<i>Allen Cole, Univ. of Utah</i>	

UUCP Mapping Project

Fri-1B (9:00 – 10:30 am)

Chair: Mark R.Horton, AT&T Bell Labs

What is a Domain?	368
<i>Mark R.Horton, AT&T Bell Laboratories</i>	
Proposal for a UUCP/Usenet Registry Host	373
<i>Mark R.Horton, Karen Summers-Horton, and Berry Kercheval, UUCP Project</i>	
A Reliable Mail Service for the UUCP Net: Implementation Status Report	374
<i>Berry Kercheval, Zehntel, Inc.</i>	
UUCP Site Name Registry <i>Lauren Weinstein</i>	
Unknown Mailer Error 101, or Why Its So Hard To See You <i>Scott Bradner, Harvard University</i>	
Panel Discussion <i>UUCP Project members</i>	

4.2BSD Panel and Q & A

Fri-2B (11:00 – 12:30 pm)

Moderator: Kirk McKusick, U.C. Berkeley

What is the Future for 4.2BSD?	375
<i>Panelists include Mike Karels (U.C. Berkeley), Sam Leffler (Lucasfilms), Robert Elz (University of Melbourne), Bill Joy & Bill Shannon (Sun Microsystems).</i>	
Exhibitor Information	377
<i>In alphabetical order.</i>	
Author Index	388

Towards a UNIX Standard

 Michael Tilson
 Human Computing Resources Corporation
 10 St. Mary Street
 Toronto, Canada, M4Y 1P9
 416-922-1937
 {decvax,utzoo,utcsrgv}!hcr!hcrvax!mike

The great flexibility of the UNIX system together with the peculiar history of AT&T releases have combined to produce a number of somewhat incompatible commercial and academic versions of UNIX. From the chaotic multitude of UNIX versions a demand for standardization has arisen. This demand also coincides with current AT&T product marketing directions.

However, the impetus towards standardization has created a sense of unease in the original UNIX community. Support and marketing groups at AT&T now seem to control the future direction, rather than the perhaps more tasteful research groups. There is a fear that innovation will be stifled. As a result, standardization proposals are met with some resistance, particularly standardization which may appear to be based upon well financed marketing efforts rather than technical excellence.

Despite the dangers, there are good reasons to believe that standards will prepare the groundwork for the next wave of innovation, and that benefits of a standard far exceed the costs.

UNIX History

Since UNIX is the product of a single vendor (AT&T), one might think that a standard should not be necessary. However, the history of the system has caused a large degree of divergence. It is important to review the features of this history in order to understand the background of current standardization efforts. The following history is somewhat simplified, but adequate for the purpose. (Those familiar with the history of UNIX versions may skip this section.)

The UNIX system was developed by Ken Thompson and Dennis Ritchie of Bell Laboratories as a personal tool for program development using only the most limited hardware environment. The system was a synthesis of many existing good ideas, and a few new ones. Because the basic conception was the product of only two minds, the early UNIX system was especially elegant and exhibited a unity of design rarely found in larger-scale systems.

This system was used internally at Bell Labs, and was gradually improved. It came to the attention of the academic community, and was released at no charge and totally without support for educational purposes. The first release that had any significant distribution was the "Fifth Edition", now known as "Version 5". (At that time, there were no UNIX "versions". Instead, the Programmer's Manual was updated with a new edition as the system changed. With each new edition of the manual, a tape copy of the research computer system disk was made. This essentially became the distribution tape.)

By the time of the first outside release, the system had been coded in the "C" programming language, but it ran only on the DEC PDP-11 processor.

The "Sixth Edition" ("Version 6") had a fuller set of functions. With this version, AT&T allowed commercial users to purchase source copies of the system. The price was fairly high (\$20,000), and the system came "as-is", with absolutely no support. It still ran on the PDP-11, and only a limited number of configurations were supported. Version 6 spread rapidly in some academic circles, and began to be used in commercial and government projects.

Version 7 of UNIX provided a fuller set of features. In addition, much of the system had been re-designed to allow portability. Internally within Bell Labs the system had been re-targeted ("ported") to another processor type. AT&T started to realize that the system had commercial potential, and allowed for binary redistribution of the system by independent vendors at a lower price. Still, no customer support was provided.

UNIX 32V was the only "port" of Version 7 to be released. It supported the 32-bit DEC Vax architecture. Otherwise, it was nearly unchanged Version 7. Only a very limited number of Vax configurations were supported.

Version 7 was the last version of UNIX ever to be released to the outside world by the original Bell Labs "research" group. There is a Version 8 internally, but it is unobtainable. Thompson and Ritchie are still associated at times with UNIX system development, but only within this research area. Version 7 was the basis of all of the early "commercial" versions of UNIX. The system was ported to various microprocessor types by a number of companies, and offered with commercial support in binary form.

After Version 6, the first of many splits in the line of UNIX development occurred. Internal Bell System users wanted UNIX, but needed a more solid "commercial" product. For this reason, Western Electric produced the "Programmer's Workbench" or "PWB" system. This ran on the PDP-11, and included a number of programmer productivity tools, as well as much more complete support for a variety of PDP-11 hardware. PWB was essentially an internal Bell System supported product. It was available to the external world, again with no support. The PWB line of development is the basis of the "USG" ("UNIX Support Group") systems, and may perhaps be considered the direct

ancestor of all other UNIX systems available from AT&T.

All of these releases were made years after each version had been made available internally to the Bell System.

UNIX spread rapidly in the academic world. The system was intellectually elegant, easy to use for research purposes, free, simple, and available in source code form so that changes needed for computer science research purposes could be accommodated. Most universities had UNIX machines, which were typically run by unpaid student "UNIX gurus". The system was easy to change, had no standards, and had no support contracts which might become invalid. The result was a torrent of changes and features, some useful, many not. Early meetings of what was to become Usenix concentrated on these changes, and many tapes were traded back and forth.

When UNIX 32V was released, a number of academic institutions were upgrading from 16 to 32 bit computers. 32V was a very rough release. It did not support many Vax configurations. It was difficult to install. It was oriented towards older hardware (since the release was two years old). Certain productivity features were missing, such as screen editing. The demand paging ability of the Vax was not used. The University of California at Berkeley ("UCB") set out to rectify these problems. The result was a series of UNIX releases from UCB, called "Berkeley Software Distributions". Release numbers are in the form "4.xBSD". The UCB effort resulted in increased performance, greater hardware support, demand paging, useful features, and a myriad of additions, new options, experimental ideas, and simply gratuitous changes. UCB included many of the features that had been added elsewhere, and seemed to make an effort to provide as many programs as possible on its distribution tapes, which were made available for a handling charge to any UNIX source licensee. The most widely used version was 4.1BSD. Most Vax UNIX sites used this system in preference to 32V, since it had higher performance, and was likely to boot on a typical Vax configuration without any custom modifications. 32V on the other hand could only boot on a specific limited set of configurations.

UCB had obtained funding to meet the UNIX needs of Darpa (Defense Advanced Research Projects Agency). These needs included high throughput image processing, and support of Darpa standards for local and long haul networks. At the same time, UCB set out to "fix" many things that were "wrong" or "missing" in UNIX. The culmination of this effort is the 4.2BSD system.

In the meantime, AT&T had been busy. UNIX System III provided lower binary pricing, and most of the features of PWB and Version 7 combined. (Although released after Version 7, it was part of the USG line of descent, and it in fact did not include some V7 features.) Both the PDP-11 and Vax were supported. This was touted as the first "commercial" UNIX offering. However, the distribution was hurriedly packaged, and the documentation reached a new low of inconsistency. For the first time, AT&T started to actively market the system. AT&T marketing created a demand for "System III". Most commercial vendors

added the important additional System III functions, but few gave up their already ported and reliable Version 7 bases. Commercial systems became Version 7/System III hybrids. Most commercial vendors also included Berkeley functions. All vendors sold software under System III licensing provisions in order to obtain the lowest price, even if the actual system sold was based on an earlier UNIX version. (System III licensing included a license for all previous versions as a subset.)

System V followed System III. With System V, AT&T brought the outside world up to date with the internal USG version. (System IV was skipped.) The system was cleaned up, and the documentation put into better order. Certain Berkeley features were added, and performance issues were addressed. Full scale marketing commenced. Full commercial support was provided to source customers. New features were added.

System V was largely upward compatible with System III. Previously, successive releases had been seriously incompatible. System V Release 2 is now available. It represents a truly upward compatible addition to System V.

Today, there are a variety of commercial UNIX systems. Most commercial vendors are moving towards System V. However, successive releases, hybrids, upgrades, offspring, proprietary enhancements, and vendor subsetting have resulted in a situation today in which end-users are unsure about what they are purchasing when they buy "UNIX".

In the academic world, many sites are moving to 4.2BSD at the moment. This is a natural outgrowth of the fact that Vaxes are the academic computer of choice, and BSD systems were for a long time the only available option to support a variety of Vax configurations. However, the BSD stream has become quite divergent from the USG stream, creating many conversion and compatibility problems. It is now misleading to the uninitiated to have both systems called "UNIX". This has been a source of a lot of confusion.

Despite the variations, it is important to remember that all of the systems, even 4.2BSD, remain recognizably derived from "UNIX".

Emerging UNIX Standards

The UNIX system has many advantages which explain its popularity: It is multi-process, multi-user, powerful, elegant, and portable. Here we will focus on portability and standardization.

The UNIX system is highly (although not perfectly) portable. UNIX implementations exist (or are in progress) on a multitude of machines with varying architecture. A related feature is the system's span of usability. A UNIX programmer can use anything from a cheap desktop micro to a giant supercomputer. To a large extent, the procedures used by the UNIX programmer will be exactly identical on each machine. There is no other available system which can accommodate a factor of

10,000 difference in machine power. The same applications programs can be used.

As we have seen, this portability is marred inconsistency and lack of standards. Commercial UNIX systems vary unnecessarily from one vendor or machine to another. There are few standards for command options, error messages, etc. It should also be noted that there is no binary compatibility. Applications software vendors can not produce one UNIX binary for each CPU type (e.g. Motorola 68000). This fragments and inhibits the applications software market, to the detriment of vendors, and more importantly, to the detriment of users wishing to purchase to best available, lowest cost software.

UNIX should be viewed as a vehicle for applications programs. UNIX systems provide an applications software architecture which is independent of machine details, including processor instruction sets, memory management schemes, and I/O architecture. UNIX is a "software bus", which allows applications to move without change. It should not be necessary for the typical application to use any system function which is variable from system to system. In particular, this means there must be a minimum set of functions which are guaranteed to exist on all implementations, and guaranteed to have the same specifications on all implementations. If these criteria are satisfied, application software can be developed and tested once. From that point on, it can be moved from environment to environment without further attention.

The unnecessary variability of UNIX systems hinders the growth of UNIX usage. The system buyer would like to know that it is enough to specify "UNIX" in a purchase order, without having to ask questions like "Does it have awk?" or "Are FIFO files implemented?". These questions are irrelevant to the end user of applications software, even though the answers are highly relevant to whether the software can actually run.

A software standard is a treaty between producers and users of a system. From the producer or supplier end, a standard specifies the functions which must be present. Other functions can be supplied, but the standard specifies those that must always be supplied, and how these should work. From the user end, the standard specifies the only functions that can be assumed if software is to be portable to all standard-conforming systems. In this paper the "producer" is the supplier of the systems software. The "user" is the writer of application software, either a software vendor or an end user. From the "applications software bus" point of view, a standard defines an exact meeting point, a minimum for suppliers and a maximum for users.

Although this is the most important area for initial standardization, there can be many levels of "producers" and "users" involved. For example, the authors of the systems software are in turn consumers or users of device hardware. Each level of producer/user relationship can be the topic of a standard. It is advantageous to prevent intermingling of the levels in one standards document, unless each level is clearly defined.

Standards promote stability and reusability of software. A system standard decouples decisions made by applications software producers from decisions made by the system producer. The standard allows each to conduct business in an orderly way, with a minimum of conflict. The purpose of a software standard is inherently related to portability, either from machine to machine or in time -- from version to version. The portability issue is what distinguishes a standard from a mere product specification.

Standards are introduced for economic reasons. One can always re-write software for different systems, but it is much more economic to avoid this. Once a standard becomes widely accepted, the economic impact of incompatible change becomes so large that change is almost unthinkable. As a result, there is inherent conflict between innovation and standardization.

Due to the unchanging nature of a standard and the large potential economic consequences, the most important standardization decisions have to do with what is left out, rather than with what is included. Once something is in, it is in forever. Some inclusions are less dangerous than others. A specification that a particular UNIX command is to be included is not terribly dangerous. If a better command comes along, the old one can still be supplied. User manuals can indicate the commands which are desirable, as opposed to those which are included solely for standard conformance. On the other hand, some standardized items rule out other solutions. For example, if the syntax of file names is completely specified, other naming conventions are forbidden. Here great care must be exercised.

Despite its variability, the UNIX system is already widely viewed as a standard. There is general industry demand for standardization. This demand is being encouraged by AT&T. The /usr/group Standard represents one result of industry demand. Other members of the industry are promoting UNIX software standardization. A recent example is the "ISIS" standardization effort proposed by several applications software vendors.

The new "commercial" AT&T recognizes the need to regain control of its UNIX product. Therefore, AT&T is heavily promoting sales and support of its latest UNIX release, System V. There is a commitment to upward-compatibility in future releases. AT&T has arranged that all four major microcomputer chip makers will support a standard certified version of System V. Few will have missed the massive advertising campaign, carrying the message in publications ranging from the Wall Street Journal and Scientific American to the computer trade press. The message is: System V is the one true UNIX, and the only version worth having. Of course, this message is not technically true, but the marketing campaign is having a strong effect, especially since it fits directly with customer desires for standardization.

In the area of UNIX standards, the following appear likely:

- o UNIX System V will form the basis of the standards.

- o AT&T will back standardization, and will maintain its commitment to upward compatibility.
- o The /usr/group Standard will be formally adopted in 1984.
- o Vendors and purchasers in the business market will demand standard-conforming systems. In particular, government purchasing requirements will specify standard systems, initially by specifying AT&T standards, and later by using the government's own standards or formal industry standards.
- o ANSI and international standards organizations will, in the longer term, adopt a standard. (Note: ANSI X3J11 is working on the C language standard, which will include many standard UNIX library routines.)
- o Academic usage will converge towards standardization over time as conforming systems become more functional.
- o UNIX will become truly generic -- the industry standard interface between applications software and computer systems. (Note: as a result, AT&T may eventually have trouble protecting both the UNIX "trade secret" and the UNIX trademark. Effective standards will also deter arbitrary change by AT&T itself, for example, changes intended only to promote sale of "3B" hardware.)

The /usr/group Standard

The /usr/group Standards Committee has been active since 1981. The committee has wide representation from the UNIX industry. Committee members serve as individuals, but the voting members include representatives from approximately 40 organizations, including AT&T.

The /usr/group Standard is a semi-formal document. It has not gone through the treatment given by an international standards body such as ISO. Nevertheless, the Committee has given a great deal of consideration to a number of issues. The purpose of the Standard was to provide the set of features needed by the vast majority of applications software, at the systems interface (i.e. C subroutine call) level. Although stated to be derived from System III, the current document is essentially also a subset of System V, with greater precision, and with one extension. The Standard was not intended as a "UNIX Standard", but rather as an operating systems interface standard derived from UNIX and UNIX-like systems. (In practice, however, it is a UNIX standard.)

The final draft standard was approved by an overwhelming roll-call vote at a Committee meeting held concurrent with the January 1984 UniForum conference in Washington, D.C. The AT&T representatives voted for approval of the Standard. At this writing, the document is before the voting members of /usr/group for final approval.

The main areas of debate were:

- o Size. Some members wished to have a more minimal set, while others wished to include as much of UNIX as possible.
- o Change. Committee members often wished to repair perceived problems in UNIX, or to make the system "better". To a large extent, this was resisted.
- o Precision vs. speed. The Committee felt that it had to complete its work in a timely fashion. The Standard is more informal and less precise than some other standard documents. This decision probably cut two years off the publication time.
- o Conformance with current practice, backward compatibility. Some redundant items are included in the standard (e.g. both "dup" and "fcntl" system calls), in order to preserve the large body of existing applications programs.
- o Implementation dependencies. The committee tried to define the standard so that it could be implemented without standardizing undocumented idiosyncrasies of the Bell Labs code. UNIX "look-alike" vendors lobbied for changes to accommodate various "improvements" which existed in those vendor products. Other vendors lobbied against deviation from AT&T.
- o Extensions. Various extensions to the system have been proposed. Only the "record locking" extension was accepted. (Note: we are informed that a future release of UNIX from AT&T will contain this feature as an interface to a somewhat larger facility.)
- o UCB compatibility. Two members of the Committee argued against final adoption of the Standard on the grounds that it may be difficult to support under 4.2BSD, and in particular that standard features such as FIFO files created problems in networking environments. Most members of the Committee felt that this latter issue represented a deficiency in some particular networking code, rather than a fundamental objection.

The AT&T representatives have informally indicated that AT&T intends to conform to the standard in future releases.

The Standard has some trouble spots, and work remains:

- o Terminal control is not standardized. Many vendors had different conventions derived from different UNIX versions. It is expected that a future update to the Standard will contain a System V subset, chosen so as to be reasonably easy to implement on most existing UNIX implementations.
- o Commands are not standardized. Only the "systems interface" is now defined. This makes it difficult for applications software to take advantage of the UNIX "tool building" philosophy by calling existing commands.

- o The Standard is at the source level. This is the place to start, but lack of binary compatibility is a current drawback in the UNIX market.
- o There likely remain some internal inconsistencies and fuzzy areas. These are minor, but annoying.

Dangers and Opportunities

Despite any problems, the proposed Standard represents wide industry agreement, and is a very hopeful sign in the development of the UNIX market. The availability of a standardized operating environment will dramatically heighten competition in the computer industry. As long as the chosen processor runs a compatible version of UNIX, a user can feel free to mix and match the hardware for best performance. At the low end, there are now approaching a hundred 32-bit UNIX micros to choose from. The high end choices are more limited, but a number of new companies are going after the "three times a 780" market, and the larger companies are expected to introduce machines in that range. Users will finally be free of vendor "lock-in" via proprietary systems. Price and performance will be the main issues. Compatibility means freedom.

The impetus towards standardization has created a sense of unease in the original UNIX community. UNIX development has always been characterized by a spirit of innovation, and there is a feeling that innovation will now be stifled. As a result, standardization proposals are met with some resistance. Despite the dangers, there are good reasons to believe that standards will prepare the groundwork for the next wave of innovation.

Systems evolve from an initial pure concept, through the first commercial realization, into maturity, and finally into senility. A good initial concept is usually one that synthesizes and unifies many previously disparate features, but which does not attempt to solve all of the world's problems. (For example, the one UNIX ordinary file type can conceptually replace most file types found on other systems in most applications.) As a system matures, additional features are added to fill out the functional range. However, as Rob Pike notes, the system gets five times bigger without getting five times better. Eventually, no amount of addition, hacking, or patching can be superior to the next pure synthesis. It is most useful to standardize before this point is reached, and get on with the research necessary to generate the next concept, rather than hastening the senility of the current one.

A standard operating environment will in fact provide a solid reliable basis upon which software developers can build. UNIX represents a much better standard than most. It is time to stop solving problems by changing the kernel calls or changing the meaning of options. One must recognize that, although a change may make the system better, it has to be enough better to be worth being different.

Finally, it is good to keep in mind that a "UNIX standard" does not prevent wholesale change or innovation. The UNIX system will continue to provide be a fertile ground for experimentation. A standard may prevent calling the result "UNIX", but that in itself is not a bad thing.

What Next?

The computer industry is at an historic turning point. If industry agreement on systems standards is achieved, users will be able to freely choose among a variety of hardware architectures. Programmers will no longer waste time adapting software to new environments, but instead will have more time to engage in creative work. The software industry will have a hitherto unparalleled degree of software portability.

The actions of AT&T and IBM will have a big effect on the emerging industry standards. AT&T has already released its new computer products, all of which run UNIX System V. But the biggest force is of course IBM. IBM has experienced tremendous success with a "generic" operating system on the PC. IBM is moving forward on the UNIX front, with announcements for PC/IX for the PC and Xenix for the 9002. By the time this article goes to press, there will likely be more announcements.

Will IBM conform to the AT&T standard, will it go with 4.2BSD, will it go with some other radical variant, or will it go with its own proprietary portable system? Unlike many other companies, IBM is large enough to be able to do all of this at once. Since each IBM division operates with some degree of autonomy, we should expect in the short term a little bit of everything. However, keep in mind the fact that IBM can well afford any conceivable proprietary operating systems development. In the long run, the only reason for IBM to go with UNIX is because the market demands generic systems. Since UNIX is so portable, if IBM does not supply it, somebody else will. This will increase the incentive for IBM to jump on the bandwagon.

On the other hand, if wide industry agreement on UNIX standards does not solidify quickly, IBM will perhaps be free to push forward its own proprietary software as some sort of industry standard. This possibility represents a grave threat to computer users, far worse than any inconvenience caused by minor technical flaws in a standardized version of UNIX.

ACSNET – The Australian Alternative to UUCP

Piers Dick-Lauder

Basser Dept of Computer Science
University of Sydney

R. J. Kummerfeld

Basser Dept of Computer Science
University of Sydney

Robert Elz

Dept of Computer Science
University of Melbourne

ABSTRACT

ACSNET is a network with goals to serve a function similar to that currently served by the UUCP network. Routing is implicit, and addressing absolute, with domains. The network daemons attempt to make use of full available bandwidth on whatever communication medium is used for the connection. Messages consist merely of binary information to be transmitted to a handler at the remote site. That handler then treats the message as mail, news, files, or anything else. Intermediate nodes need not consider the type of the message, nor its contents.

1. Introduction

ACSNET is a loosely coupled network of heterogeneous machines, and has a purpose and function similar to that provided by the UUCP network in wide use in most of the UNIX[†] world.

Before continuing we must make two points. First the matter of naming. Perhaps ACSNET's biggest problem is the lack of a suitable name. The developers (PD-L and RJK) call the software 'The Sydney Unix Network' or SUN for short, while the *network* built using SUN is called ACSNET. Unfortunately,

[†] UNIX is a Trademark of Bell Laboratories.

'SUN' may be confused with a Unix based workstation of the same name and 'ACSNET' suggests some relationship, or at least similarity with the U.S. CSNET network. No other label has gained sufficient approval to catch hold, so ACSNET serves for this paper.

Second, a note on our use of UUCP for comparisons with ACSNET in this paper. The authors have no intention to discredit UUCP, or to belittle its achievement in linking the world's UNIX systems in a manner never before attempted. However, its presence as a current *de-facto* standard for UNIX to UNIX communications places it in a position where we cannot avoid making comparisons in order to illustrate certain points with far greater economy of words than would otherwise be possible.

2. Overview

ACSNET provides a message passing service, from one host to another, possibly utilising intermediate hosts in a store and forward manner. Messages may be mail, files, printjobs, news, or almost anything that can be transferred in a string of bytes.

Routing in ACSNET is implicit, users need only be concerned about the name of the host at which the message is to be delivered. They need not be concerned about which hosts the message might visit on its journey to its final destination. If, for some reason, a message is undeliverable, the network will make every attempt to return the message to its original sender.

Messages can be transported over any medium capable of supporting a connection between two hosts. This may be phone lines, ethernet, X.25, twisted pairs, etc. Preferably, the link should provide a transparent 8 bit data link, but links with only 7 useful data bits can be accommodated (at a slight loss in throughput). The network daemons make good use of full duplex communication channels, transferring messages in both directions simultaneously, providing, in ideal conditions, effective throughputs up to twice that attainable with UUCP.

3. Messages and Handlers

A message is a string of bytes addressed to a handler at one or more hosts. A handler is a process that will receive the messages at the final destination. Typically the handler will impose some further protocol, often recognising a user name (in some form of representation) that the message is directed to. A message, of itself, is addressed merely to a host and a handler.

The notion of messages addressed to handlers is one of the primary differences between ACSNET and UUCP. UUCP functions as a remote command execution system built upon a file transfer protocol. Mail, news, etc. are transmitted by sending the content of the item to the remote system as a file, then sending a request to execute the remote mail, or news, receiver with the file as standard input. ACSNET simply transfers a message addressed to the mail handler on the remote system.

ACSNET uses trailer protocols, where the *header* follows the message. This allows file copying to be avoided on intermediate hosts when routing statistics are updated. ACSNET only ever performs disc to disc copying of a message when it is being copied to its final destination, and optionally, when queueing the message in the first instance.

Currently handlers exist for mail, news, file transfer, and remote printing. A remote command execution handler could be added if the security issues could be adequately solved. Any other handler could be created just as easily, for any purpose that the sending and receiving hosts agree upon. Unlike UUCP, it is not necessary for intermediate hosts to know of the new handler for correct functioning.

4. Addressing

All messages carry a destination host address. This is the ascii name of the destination host. Messages also contain a sender name and address, and may contain a user address. This last item may be anything that the handler requires for its functioning.

Messages may be addressed to more than one host. A copy of the message will be sent to each host addressed, and that will be done with the minimum possible message traffic (or something approaching it). A message may also be broadcast to all hosts. This is most often used for network management messages, such as new hosts connecting, and similar events.

Users also have the option to guide their message through a specific set of hosts. Primarily this is used for network testing, loopback messages would otherwise be impossible to create.

Such a route is expressed in a notation borrowed from UUCP as

host-1!host-2!host-3 ...

However, note that there is a fundamental difference between this form of addressing and UUCP addressing. Each *host-N* is the absolute address of some site. The ! does not imply a link between two adjacent hosts. In the above example, the message will visit, in order, host-1, host-2, and host-3. But the route taken to travel from host-2 to host-3 is not specified, and in fact, given a suitable topology, the message might travel that route via host-1!

Strictly, in all the above, the term 'host name' should be replaced by 'domain specification', but that is harder to type, read, say, and think about. Any of the places where a host name was specified, ACSNET would really expect a domain specification. Domains might simply be host names, or they might specify local sub-domains, or perhaps a domain that is not within the ACSNET network, in which case the message will be sent to an appropriate gateway.

Note, nowhere here has it been specified what syntax should be used by users in communicating with user agent programs. It is to be expected that

user@domain

will be the most common format, though the older Australian net syntax of

user:host

will be supported into the distant future. Almost none of the ACSNET code either knows or cares what syntax users will use to send messages.

5. Routing

Each host maintains two tables† to contain network information. The first of these is the network state file, and contains for each known host, a list of the domains to which it belongs, a list of the hosts to which it is directly linked, and the cost and current status of each of those links. One of the domains is special, and is considered to be the primary domain of the host.

† For 'table' read 'file'.

The table can also contain various other information, such as a human understandable description of each host, and statistics on messages and bytes sent, received, and passed through. This information is optional, and probably would be deleted on a small host.

The state table is considered public information, and is sent to any host that requests it. It is broadcast to all hosts whenever a new link is added.

The second table is the routing table. This table indicates which link should be used to transmit a message bound for any host or domain on the net. It is built from the state table, usually whenever that is altered. As each message arrives on a link it is passed to a routing process. That process passes it to its appropriate handler if this is (one of) its final destination(s), and queues it to be transmitted on the next link if the message has further to go. The routing table is used to make this decision.

Information on which links to transmit a broadcast packet that originated at any particular host is also retained here. This is arranged so that broadcast packets travel over a minimum traversal of the graph, and implements Dalal and MetCalfe's extended reverse path forwarding algorithm.†

This table also contains miscellaneous information, such as local aliases for hosts on the network. It is private information, and is not exported to other hosts.

Routing messages are broadcast to all hosts in the sender's primary domain whenever a link changes status (goes up or down). These are brief messages, indicating the nature of the change, and carry a timeout age, after which they are deleted wherever found. This works well, as typically, distant parts of the net are not interested in local changes, which often might have become outdated before the message reached the host. Rerouting to avoid links that are down can usually be handled by nodes relatively close to the broken link.

6. Gateways

The routing table for any local link can indicate that a non-standard spooling program should be used to send a message over a designated link, or to deliver a message to a particular domain. This can be used to build interfaces to newer, or older, versions of the network software, or to implement a gateway

† Y. K. Dalal and R. M. MetCalfe, CACM, Dec. 1978.

to a foreign network. The spooling program is responsible for performing any transformations required of the message to meet the standards required by messages entering the new network.

This performs admirably when interfacing to a network with similar capabilities, but is less of a success connecting to UUCP for anything but mail, as there is no standard way of performing possibly multi-hop file or news transfers.

7. Calls and Daemons

ACSNET uses node to node daemons to transfer messages from one host to another. Unlike the UUCP uucico process, ACSNET daemons have no knowledge of how, or when, to connect to a remote host, except in the trivial case where that is accomplished by opening a tty compatible special file. To handle other cases, the routing table may contain the name of a process to run to establish a connection to another host. That process is expected to make the connection, then exec the daemon with standard output open (read-write) to the remote host. This permits easy expansion to a wide variety of possible connection types.

A pair of daemons transfer data between themselves over three channels in each direction simultaneously. This allows up to 6 messages to be in flight between any pair of nodes at any one instant. Messages are assigned to one of the three channels based upon their size. This allows small messages to overtake larger ones on another channel, and prevents those extraordinary delays that can occur when a particularly huge message is being transferred.

The daemons also keep track of the current position in each message in transit. This allows messages to be restarted (usually) without transmitting data that had been received correctly at its destination, should a link die prematurely, or a system crash.

Messages on each channel are sent via a windowed packet scheme, similar to that used by HDLC (and UUCP, and X.25), and are checksummed using the standard CCITT CRC-16 algorithm. This checksumming can be disabled for a link if it is known to be reliable, typically if ACSNET is used over another protocol. Messages can also contain end to end checksums, to guard against corruption while waiting at an intermediate node.

8. Status

ACSNET is currently being used on VAX 11/780 and 11/750 processors, PDP-11/40 and 11/34's, Sun Workstations, Perkin-Elmer, Plexus (P60), and other less widely known machines. These processors are variously operating under V7, 4.*BSD, System III, and System V. A VMS version has been suggested, but at this stage, not attempted.

There are about 90 hosts on the Australian network, and though most of those currently use the previous software, most will probably convert in the near future.

9. Todo

There will be a message disassembly, reassembly facility, to permit huge messages to be transmitted without overloading intermediate nodes.

A UUCP gateway is needed. This is hard because of problems with multi-hop file transfers.

The code to handle domains is still in final stages of implementation as this paper is written. By the time it is presented, that should have been completed.

Some tuning remains to be done. One possible improvement on System III and V systems, and 4.2BSD, would be to use the available interprocess communication mechanisms (named pipes, sockets) to allow the routing process and handlers to become daemons, and be created just once, rather than once per message.

10. Availability

The source code will be available under license. Anyone interested should apply to Bob Kummerfeld at the address above.

11. Conclusions

ACSNET is a suitable network system for connecting comparatively large networks, of a size comparable to the UUCP network, that may operate in a relatively unmanaged environment. Its implicit routing makes it considerably easier to use than standard UUCP, and more accurate and adaptable than the heuristic UUCP routing algorithms now becoming available.

We feel that ACSNET would be suitable as a general replacement for UUCP.

Broadcasting of Netnews and Network Mail via Satellite

Lauren Weinstein
Computer/telecommunications consultant
PO BOX 2284
Culver City, CA 90231
(213) 645-7200

Over the last several years, the volume of Usenet netnews traffic being carried by dialup telephone circuits has increased far beyond that originally envisioned. Not only has the number of participating Usenet sites grown enormously, but the number of articles being submitted daily into the network has also increased dramatically. All indications point toward this growth continuing at an increasing rate.

The vast majority of existing Usenet sites transfer netnews via UUCP connections over dialup telephone connections, usually at 120 characters a second and sometimes at even lower speeds. The nature of the network is such that many relatively lengthy telephone calls are required, on the part of many sites, to successfully "flood" a given article throughout the network to all sites which might wish to receive it. The overall cost of these calls is not trivial and could well act as a deterrent to the successful continued operations of both Usenet and the UUCP network which underlies most of Usenet.

However, there is an alternative to multitudinous telephone calls for distributing netnews articles or other materials (such as mail between UUCP network backbone hub sites). Rather than sending such items by phone at low speed, satellite technologies could be employed to literally broadcast these items simultaneously to most sites in the continental United States and parts of Canada. I have begun a project to investigate the technical, economic, and "political" aspects of such an enterprise.

Technically, the "broadcast netnews" system would work as follows. An area of the conventional television signal called the "vertical blanking interval" is mostly or completely unused by most television broadcasters. This space can be used for the transmission of relatively high speed data given the proper (reasonably simple) equipment, without interfering with the primary video being transmitted by that signal. Vertical interval data space would be obtained on a contract basis from a television programming source (or sources) who already had geosynchronous satellite transponder capacity. Ideally, such sources would be widely received by numerous cable television

(CATV) systems and would not have plans for scrambling or their own exclusive uses for their vertical interval.

Netnews articles (or messages such as appropriate UUCP direct mail) would be sent via dialup lines as ordinary single-destination UUCP messages to a pair of dedicated microcomputers located at the studio or satellite uplink facility of the television programming source. These micros would multiplex the incoming articles into a continuous data stream (complete with robust error correcting codes), possibly combine them with other data, and insert the stream into the source's vertical interval. This data would normally be invisible to conventional television viewers and has a maximum practical capacity of about 50K bits/second. Such a capacity would allow for many multiplexed netnews items and other materials to be transmitted simultaneously at high speed. The pair of microcomputers, with one acting as a "hot" standby, would help to avoid the possibility of an equipment failure causing a major disruption of the network, though dialup netnews transmission between Usenet sites could still be used as an ultimate backup if necessary.

Usenet sites would receive the transmitted data via special decoders at their locations. These decoders could be connected directly to a local CATV hookup if available (assuming that the local cable company carried the transponder of interest) or could be connected to an inexpensive (approximately \$1000 or less in most cases) satellite receiver for direct reception of the desired satellite signal. The decoder would have various possible data outputs, including low speed (1200 bps) outputs which would simulate a conventional modem, and higher speed outputs for those computers which have the facilities to handle high speed data. By using generous amounts of memory within the decoders, a substantial number of typical netnews articles could be received quickly within the decoders themselves, and then output to the local computing equipment at the highest speeds such equipment could conveniently process. Since memory is now fairly inexpensive, it would seem reasonable to expect these decoders to contain at least 128K bytes (and quite possibly more) local memory for such purposes.

While some special software would be required to handle this data, the amount should be fairly minimal. A primary aspect of this software would be to track the continuous broadcast repetition of recent messages which would be used in place of end-to-end

message acknowledgement techniques in such a system. The very high speed of the broadcast data channel makes the "repetition" mode of operations both effective and highly practical for this application. In cases of sites which had been unable to receive any messages for long periods (e.g. were "down" for a prolonged time), special control messages could be directly mailed via UUCP to the broadcast micros to request special rebroadcasts of older messages still online at those micros.

The basic hardware for the decoders is all "off-the-shelf" and would make use of existing integrated circuit chip sets designed for commercial vertical interval decoding operations (such as "Teletext" services). In reasonable quantities, the decoders should cost about \$500 or possibly less. The design for the prototype decoder for this system is already underway.

While the system I've described would mainly be useful for the distribution of netnews articles for broadcast throughout the network, the same distribution system could be used for certain classes of point-to-point mail, particularly if encryption were employed. Typically, this application would be limited to the sending of mail between major UUCP hub sites which handle large volumes of traffic or from hub sites to "ordinary" sites. To be practical, the sending hub would need to be located near (in terms of phone call cost) to the satellite uplink location, and would then distribute most mail via encrypted, individually addressed messages through the satellite channel rather than via dialup lines. Dialup connections with the hub would still be required for return mail, of course, but it is possible that a substantial volume of mail traffic might ultimately be deliverable via the satellite mechanism. This sort of operation could be greatly expanded if we obtained access to multiple satellite uplinks located in widely separated areas of the country. This would facilitate the satellite transmission of hub mail traffic from multiple areas and would permit an even more complete utilization of the satellite system's large data bandwidth. While in the short run we should consider this system to mainly be designed for broadcast netnews, the possibility of future mail applications adds another dimension to the overall usefulness of the service.

While this satellite delivery system would enable many sites to receive these broadcast materials without use of dialup telephone connections, sites which chose not to fully participate in the program (or which for technical reasons were unable to participate),

would be free to continue receiving some or all items via conventional dialup or other facilities. For sites with access to the appropriate local cable television service, the cost of participation would be extremely low--little more than the cost of the decoder and the CATV hookup. Sites which needed to install a satellite receiver would initially pay somewhat more, but non-recurring costs such as the receiver would be rapidly absorbed by the large decrease in telephone costs which would normally be expected under the plan.

There are some continuing costs in operating such a satellite-based system. Primary among them are payments to the provider of the satellite service whose vertical interval is being used. From my discussions with potential vertical interval providers to date, it appears likely that reasonable cost service could be obtained, especially if the provider was able to "share" some of the data capacity of our installed data delivery system for their own purposes. Given the large data capacity of the television vertical interval, such a "sharing" concept would be perfectly reasonable and would not negatively impact our own operations. The exact cost of such services is impossible to gauge at this time, since there are many variables to be considered, but I am indeed optimistic that an affordable data carrier will ultimately be available for our use.

There are currently no plans to charge sites for participating in the satellite broadcast system. It is hoped that voluntary grants and contributions from various interested organizations will be sufficient to keep the system running without resorting to direct charging of sites. The Usenix organization, for example, has expressed interest in helping to fund ongoing costs associated with the project to help benefit the Usenet community at large. Hopefully other organizations will also contribute to the cause, in the same spirit of cooperation that has kept Usenet running up to now.

At the present time, discussions are continuing with various potential satellite service suppliers to determine the various technical factors, costs, and other issues relating to the startup of the service I've described. Software planning has already begun in parallel with the preliminary decoder design.

While such a service will not appear overnight, I have high hopes that it will begin to take complete shape in the near future. It is a fairly complex undertaking and could conceivably be derailed by various events, but it has a good chance of reaching fruition.

I strongly feel that without a system such as this to broadcast the widely distributed netnews articles and other materials, our network management problems will become extremely serious in a fairly short time and could threaten the open flow of information over the network. The satellite broadcasting plan described here offers hope of avoiding this undesirable situation while simultaneously decreasing overall network operating costs. With a little luck, it may actually come to pass!

The Berkeley Internet Name Domain Server

Douglas B. Terry, Mark Painter, David W. Riggle, and Songnian Zhou

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

ABSTRACT

The Berkeley Internet Name Domain Server allows a standard way of naming the many types of objects and resources that exist in distributed UNIX environments, and provides operations for storing and retrieving information about these objects. BIND servers collectively manage a hierarchical name space that is partitioned into domains reflecting administrative entities. Many existing UNIX applications, particularly mail facilities, will benefit greatly from such a service.

1. INTRODUCTION

Computers running the UNIX operating system are no longer small, stand-alone time sharing systems, but exist as part of larger distributed computing communities. For instance, environments in which a moderate number of computers running Berkeley UNIX are connected by a local internet are becoming quite common, especially within universities. Desires to share information with others outside of our local environments, typically through electronic mail, are also quite prevalent. The latest version of Berkeley UNIX, 4.2 BSD, has incorporated facilities to support distributed applications, such as network communication protocols, into the kernel.² Current efforts are underway in the Computer Systems Research Group at U. C. Berkeley to evolve Berkeley UNIX into a more transparent distributed operating system.¹

In order to accomplish transparent sharing of objects and resources in a distributed environment, a uniform means of naming and locating the different types of resources must be established. The Berkeley Internet Name Domain (BIND) Server is being implemented to provide such a service to Berkeley UNIX users. BIND Servers running on various machines collectively manage a database of information about named objects such as host addresses, user mailboxes, and server ports. The goal is to achieve a more transparent and less troublesome distributed computing environment.

This work was partially sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, and monitored by the Naval Electronics Systems Command under Contract No. N00039-C-0235.

2. DESIGN OVERVIEW

The BIND servers maintain a tree-structured name space, complying with the DARPA Internet Naming Convention,^{5,6} in which each node of the tree has an associated label. The name of a *domain*, or subpart of the tree, is simply the concatenation of all the labels of the domains from the root to the top node of the domain, listed from right to left and separated by dots. The labels need only be unique within the same domain. No limitation exists on the number of levels of the domain space and the number of subdomains that a domain may have.

The authority for managing parts of the name space could potentially be delegated at every domain. Specifically, the whole space is partitioned into a number of areas called *zones* that start at a domain and extend down to the leaf nodes or to domains where other zones start. Zones usually represent human administrative boundaries and associated authorities. For example, Berkeley may have a zone "ucb.arpa" and the computing center at Berkeley may have a zone "cc.ucb.arpa" under "ucb.arpa", each being maintained independently.

Servers store information about objects and resources in *resource records* consisting of a domain name, class, type and data fields.³ Each domain name may have a number of resource records associated with it. The value of the type field, along with the class, specify the format of the data field. For instance, resource records for host addresses have a type value "A" and those for user mailboxes have a type value "MB". The set of allowed types is well defined; a complete list of all the currently defined resource record types can be found in RFC883.⁴

BIND Servers consist of facilities for database management and mechanisms for managing the name space in cooperation with other name servers. The database management subsystem provides utilities for storing and retrieving resource records for a single server. The name management subsystem is responsible for adding semantics to the database, such as recognizing aliases or answering queries, as well as maintaining consistency among replicated data. The interface to the BIND Servers is through *resolvers*, a group of subroutines that users call to access the name servers. In processing user queries, for instance, resolvers are responsible for locating a server with the desired authoritative information. The relationships between the resolvers, the databases, and the name management facilities are depicted in Figure 1. These three aspects of the naming service are discussed in more detail in the next three sections. Section 6 then discusses some possible uses for BIND Servers in a distributed UNIX environment.

3. DISTRIBUTED NAME MANAGEMENT AND QUERY PROCESSING

Each BIND Server is responsible for managing parts of the name space, called zones. A general mapping between zones and servers exists, that is, a zone may be stored at one or more servers, and a server may contain zero or more zones. A name server containing a zone is said to be an *authoritative name server* for that zone. To enhance performance and availability, each ~~zone is~~ zone is generally stored in several servers; the degree of replication depends on the frequency of its use and its importance for network operation. The zone at the root of the domain tree, for instance, may be highly replicated to avoid frequent remote queries and to enable operation to continue when servers fail.

Authoritative servers are classified into a *primary name server* and *secondary name servers* for each zone. The primary name server stores the truly authoritative copy of the zone database

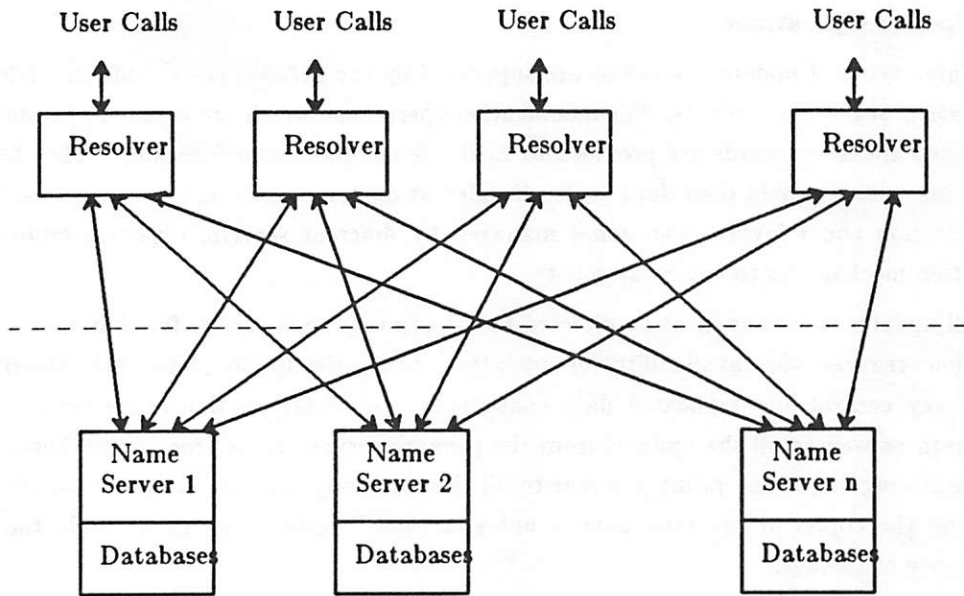


Figure 1. Relationship between resolvers, name servers, and databases.

whereas the other servers get their zone information via a zone transfer operation from the primary server at system startup time. Note that the terms primary and secondary are meaningful only with respect to a particular zone. While a server is secondary for one zone, it may well be primary for another.

3.1. Retrieval Queries

The first step in processing a client's request for information about a particular domain name, deciding which of possibly many zones to start searching, is accomplished by comparing the domain name in question with the top domain of each zone maintained by the queried server. Assuming zones do not overlap, the zone with the closest match is the only one on this server that may contain the domain being sought. For example, suppose the requested domain is "d.c.b.a", and the name server contains zones with origins "a", "e.c.b.a", and "b.a", then the zone "b.a" is selected. In this example, zone "a" delegated part of its authority to zone "b.a" and zone "e.c.b.a" borders the domain being sought.

Once the closest zone has been selected, the server must search down the path from the zone origin to the domain being sought to check if authority has been delegated to another zone at some domain along the path. If the current name server is determined to be authoritative for the ~~queried domain~~, then the requested resource record(s) are retrieved from the zone database and returned to the resolver if found. On the other hand, if the selected zone ends before the desired domain is reached, indicating that authority had been delegated to a different zone, the domain names and addresses of the servers to whom authority was delegated are returned. If no reasonable zone to search resides at the server, information about the authoritative servers for the root domain will be returned to the resolver to allow it to continue its search for the desired

domain.

3.2. Update Operations

Three types of update operations are supported by the BIND servers: addition, deletion, and modification of resource records. For modification operations, which are meant to be atomic, both the old and the new records are provided to facilitate complete error checking. The old and new records may differ only in their data fields. If different domain names and/or classes were allowed, a modification could involve two zones managed by different servers, requiring more elaborate transaction mechanisms to ensure atomicity.

All updates to a zone must be directed to the primary name server for that zone. While this restriction reduces the availability of updates, it drastically simplifies the algorithms for concurrency control and replicated data consistency. Secondary authoritative servers get their initial data as well as all the updates from the primary server. Data propagation has a radiation pattern, flowing out of the primary server to all the secondary servers. Absolute data consistency among all the copies of the zone data is not guaranteed; instead, we ensure only the eventual convergence of the data.

Incremental refresh operations, instead of whole zone transfers, are used to propagate updates after the initial zone transfer. When an update arrives at a primary name server, in addition to performing the update to its database, the server also records the update in a data structure called the *update list*. Each secondary server periodically establishes a communication connection with the primary server and sends a refresh query identifying itself and the zone desired. The primary server then responds with all of the updates since the last refresh for this secondary server. Pointers into the update list are maintained for each secondary server to keep track of how much of the update list this server has received; the storage occupied by update records that have been distributed to all the secondary servers is reclaimed.

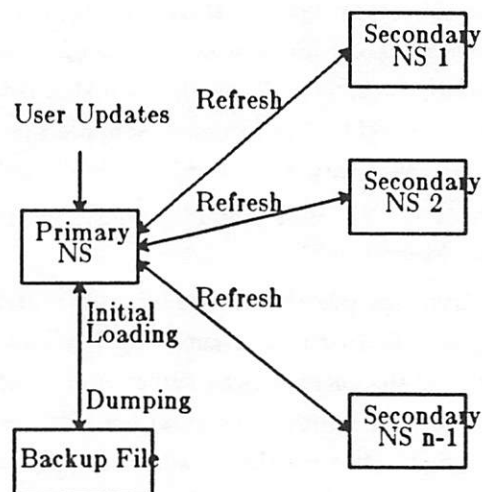


Figure 2. Name server structure in relation to a particular zone.

Figure 2 shows the name servers' structure in relation to a particular zone, and the data propagation pattern. The incremental refresh scheme has two major performance advantages. One is that whole zone transfers are avoided, thereby reducing network and host load significantly. The other is that the updates are propagated in batches, and their frequencies are controlled by each secondary server so that frequent connection costs are avoided. These advantages are particularly noticeable for servers connected over dialup lines.

3.3. Performance and Protection Issues

If name server operations are processed serially then a client's response time could be seriously impaired since a zone refresh may take a tremendous amount of time, especially an initial zone transfer. This involves requesting a connection, sending out the query, waiting for the response, and performing all the updates in the response. In the mean time, all the incoming queries are queued up. For performance reasons, a name server should be able to answer user queries and perform maintenance operations, such as zone refreshes, concurrently. Concurrent operations would require that multiple processes share zone databases and name server data structures. For example, the user query process could perform user updates and store them in the update list to be used later by the maintenance query process for propagation. Unfortunately, such sharing of information is very difficult to implement in Berkeley UNIX since each process has its own private address space and no memory sharing between processes is possible. To avoid expensive communication between processes, either through shared files or by messages, a BIND server runs as a single process; the various activities are multiplexed within that process.

Lastly, permitting interactive user updates necessitates authentication and access control mechanisms in order for the name servers to be usable in most environments. Although no access controls have been implemented in the current version of the BIND servers, access lists could easily be stored in the BIND server database to prevent unauthorized operations. Facilities for remote authentication should probably be provided by separate authentication servers.

4. DATABASE FACILITIES

The use of a large general purpose database system to support a name server would be unnecessarily expensive since only simple retrieval and update operations are required. Mechanisms for joins, selects, and even elaborate crash recovery would be superfluous. Hence, special purpose data manipulation facilities were built with simplicity and speed as the major goals.

An important design decision was to keep the entire database resident in a process' virtual address space. The amount of data stored by a BIND Server is small enough to make a memory resident database feasible on a VAX. The complete database is stored in a collection of fixed-sized buffers making re-use of storage very easy, i/o and memory operations uniform, and debugging easier. Data that is longer than one full buffer, currently 32 bytes, can be stored in several buffers which are then linked together. Two pointers provided in each buffer allow hierarchical structures to be built. Since buffers are dynamically allocated in sequential blocks of a thousand, traversing lists built from fresh buffers should not result in many page faults. A garbage collector periodically reclaims unused buffers and groups them on a free list.

A hash table maps full domain names (treated as a flat name space) into a hierarchical class and type structure built from the single sized data buffers (see Figure 3). Hash table collisions are resolved by chaining. Associated with each domain name is a zone name and a list of classes, where each class has a list of types and each type has a list of data with its time-to-live (TTL).

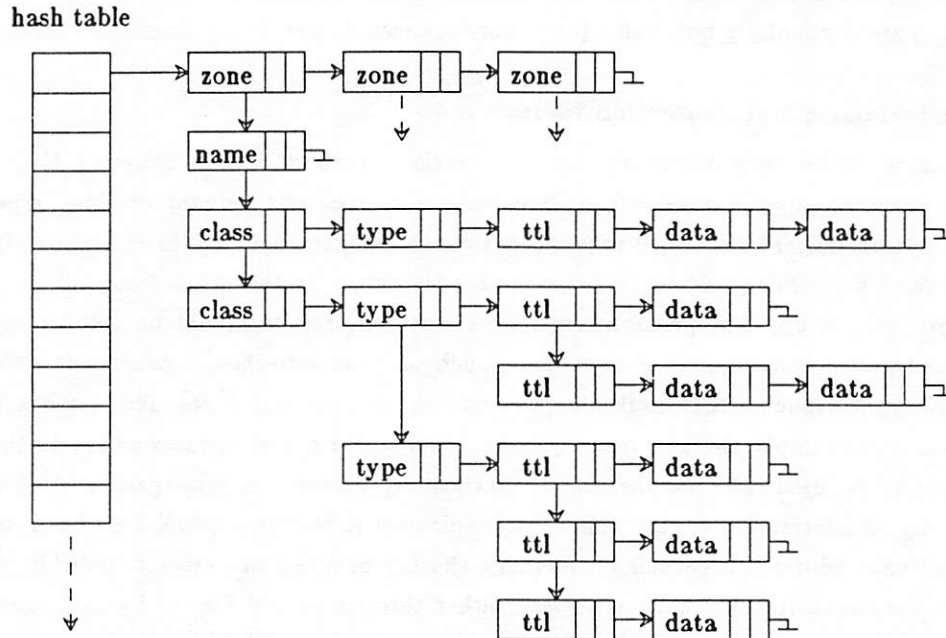


Figure 3. A sample domain name database entry.

The database management subsystem answers a standard query for a particular domain name as follows: the domain name is hashed and the collision chain examined until a match for both domain name and zone is found; the desired class is found by a linear search through the class list. From that class the search continues down its type list for the desired type. After the type is located, the data fields below it are returned to the name server process.

To survive crashes, a parallel version of the database is maintained reliably on disk. Every time a data buffer is modified, its disk buffer is similarly modified. Updates are always performed by switching a single pointer from the old data buffers to the new data buffers to render the operation atomic. Unfortunately, the UNIX operating system makes it difficult to force data onto the disk. All update routines call *flush* before returning, but no more assurance of the data actually being written to disk is provided. A consistency checker is run on the database at system startup time to guarantee that the database is in a consistent state.

5. RESOLVERS

A resolver insulates users from much of the complexity of communicating with BIND servers. Specifically, a resolver handles the details of transmitting a user's query to a name server, including the retransmission of lost queries or responses and the recovery from duplicate, or otherwise stale, responses. Resolvers use an iterative process to locate an authoritative server for a given domain. If a queried BIND server is unable to provide the desired information about the particular domain, the response contains a referral to another name server, or an error indication. Eventually, either the query will be satisfied, or the resolver will be unable to proceed.

Initially, the resolver obtains the address of at least one name server from a configuration file.

The current resolver used with the BIND servers may be accessed from C language programs as a collection of subroutines. These user level routines, whose semantics reflect the update and retrieval operations supported within BIND servers, are listed in the Table 1. The present implementation required no modifications to the Berkeley UNIX 4.2 BSD kernel and has the advantage that the cost of communicating with the resolver is simply that of a subroutine call. However, a process using a resolver cannot benefit from data that has been retrieved and cached by another process. Alternatively, the resolver's functions could be placed in the kernel so that a shared cache can be maintained.

Routine Name	Purpose
std_query	User routine for forming general standard queries.
dn_in_addr	Convert a domain name to the Berkeley UNIX 4.2 BSD structure which is used to manipulate file descriptors for IP communications objects.
inv_query	User routine for forming general inverse queries.
in_to_dname	Convert Berkeley UNIX 4.2 BSD structure which is used to manipulate file descriptors for IP communications objects to a set of corresponding domain names.
com_query	User routine for forming general completion queries.
dnc_in_addr	Identical to "dn_in_addr" except that the domain name need not be completely specified.
answer_i	Return an individual resource record from the set of resource records which are returned by "std_query", "com_query", "inv_query" or "in_to_dname".
add_rr	Ensure that a resource record is in the database of the primary name server for its domain name.
delete_rr	Ensure that a resource record is not in the database of the primary name server for its domain name.
modify_rr	Ensure that the new version of a resource record is in the database of the primary name server for its domain name, and the old version is not. (fails if neither present)
set_resopt	Sets and clears a number of options for the resolver.
set_domain	Sets the domain name of the name server to which inverse and completion queries will be directed.

Table 1. Current resolver interface

6. UNIX APPLICATIONS

One can envision many possible uses of a name server in the Berkeley UNIX environment. The most obvious use is in support of mail facilities. The procedure of editing `/usr/lib/aliases` and waiting several minutes for *newaliases* to massage the file is exceedingly annoying. If stored in a BIND Server, this information about aliases and distribution lists would be incrementally updatable and more readily available. An alias could simply be a domain name with a resource record of type "ALIAS" whose data field contained the recipient's standard name. Distribution lists could either be stored in a single resource record or as a collection of resource records, one for each member of the group. Other information could be stored with the distribution list (in a separate type), such as a description of the list's purpose (e.g. the "Doctor Who mailing list at Berkeley") or the person in charge of maintaining it.

Unix users often have an entry in the file `/usr/lib/aliases` on most, if not all, machines in their local environment that directs mail to the person's "home" machine. Ideally, mail should be addressed independent of mailbox locations with the BIND Server mapping users to mailbox sites. For example, mail could be sent to "terry@Berkeley.ARPA" instead of "terry@ucbarpa.Berkeley.ARPA" and get properly forwarded to the "ucbarpa" machine by consulting a BIND server that is authoritative for the "Berkeley.ARPA" domain. Not only would this relieve senders from having to remember and specify machines, but it would also give recipients convenient control over where their mail is received since the name server could simply be updated when a user migrates to a new machine.

Gains in managing a distributed computing environment can also be made by storing information such as the `/etc/hosts` file, the *finger* database, and possibly the password file in BIND servers. Typically this information is replicated on all machines that are under a common administrative authority; often updates must be manually performed on each machine. With BIND servers, the routines *gethostent*, *getnetent*, *getprotoent*, *getservent*, etc. would be simple name server queries. This is only a sample of the range of possible uses of BIND Servers in a local distributed computing environment based on Berkeley UNIX.

7. SUMMARY

The Berkeley Internet Name Domain Server has been designed and implemented to serve the needs of distributed computing communities. It allows a standard way of naming the many types of objects and resources that exist in such an environment, and provides operations for storing and retrieving information about these objects. A number of important decisions have been made in the design of the BIND servers. These include the incorporation of interactive user update queries, the distinction between primary and secondary name servers, the management scheme for replicated zones, and the protocol for incremental zone refresh. Emphasis was placed on being able to incorporate hosts of various types into the distributed community, including those connected over slow speed telephone lines or under different administrative authorities.

The current version of the BIND server, as described in this paper, is written in the C programming language and runs on 4.2 BSD UNIX. Applications that make use of the service must now be implemented to test out its design. Several existing UNIX utilities could easily be converted to using the resolver routines, and many new applications should be developed to

exploit the uniform distributed name space.

References

1. D. Ferrari, "The Evolution of Berkeley UNIX," *Proceedings COMPCON Spring '84*, pp. 502-505, February-March 1984.
2. W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, D. Mosher, "4.2BSD System Manual," Technical Report 5, Computer Systems Research Group, University of California, Berkeley, Draft of September 1, 1982.
3. P. Mockapetris, "Domain Names - Concepts and Facilities," RFC 882, USC/Information Sciences Institute, November 1983.
4. P. Mockapetris, "Domain Names - Implementation and Specification," RFC 883, USC/Information Sciences Institute, November 1983.
5. J. Postel, "Computer Mail Meeting Notes," RFC 805, USC/Information Sciences Institute, February 1982.
6. Z. Su J. Postel, "The Domain Naming Convention for Internet User Applications," RFC 819, Network Information Center, SRI International, August 1982.

MMDFII: A Technical Review

Douglas P. Kingston III

Ballistic Research Lab
Aberdeen Proving Grounds, Maryland 21005
<dpk@brl>

ABSTRACT

The Multi-channel Memo Distribution Facility (MMDF) is one of the most sophisticated mail systems available for the UNIX[†] operating system. MMDF is a mail transport system that supports a variety of user interfaces and delivery mechanisms. The design was not encumbered with the need to be compatible with existing mail systems, and as a result MMDF has a unified family of mail handling programs. This review will discuss MMDF's design and operation, concentrating on those features that are unique to MMDFII, the latest release of MMDF.

MMDF's design allows it to grow from a single-host system to a large mail relay without degradation of mail system performance, and to degrade gracefully as the load becomes huge. The demands of a high volume mail relay have led to many of MMDF's innovative design choices.

Unlike some other systems, MMDF has separate processes for mail submission and delivery. Recent changes to the delivery software to permit intelligent retry strategies based on the retry history for each dead host will be explained. The effect of the new domain server mechanism on address validation will be discussed.

The separation of mail into channels is key to MMDF's ability to handle large amounts of mail. Each channel represents a different class of delivery and each channel has its own queue. This isolates problems and allows one to provide a different "levels of service" to different channels.

Other topics to be discussed will include available user interfaces, the mailing list processor, aliasing, runtime configuration, and domain based naming.

The MMDF system was originally developed at the University of Delaware and has since seen significant development work at the Ballistic Research Laboratory and University College London.

Introduction and History

The Multi-channel Memo Distribution Facility, commonly called MMDF, is a suite of software that has seen a great deal of work since it was originally released in 1980. The original code was designed and implemented by Dave Crocker working under professor David Farber at the University of Delaware (UDEL). The MMDF system was then chosen to form the initial backbone software for the CSNET project and has been in use for several years by elements of the U.S. Army. The software has seen a great deal of change in the process. The original code is commonly referred to as MMDFI or MMDF Version 1. A number of minor additions and changes

[†] Unix is a trademark of Bell Laboratories.

were made while fielding MMDFI as the result of collaboration between UDEL and BRL and some other sites. After the original code was fielded in CSNET, Dave Crocker began the development of a upgraded version of the MMDF system which was designed to work in the new Internet domain naming system and was to incorporate numerous design changes suggested by experience with MMDFI. Dave Crocker left the CSNET project before completing this work, approximately two weeks before the TCP/IP switchover of the ARPANET, 1 January 1983. At this time BRL was a solid MMDF site so we were reluctant to try to retrofit the existing version of MMDFI to handle the new mail protocols that also took effect 1 January, so Doug Kingston of BRL undertook the task of finishing the work needed to make MMDFII operational. A production version of MMDFII was up in the third week of January 1983 serving as BRL's mail system on three hosts, but not until June 1983 was there a stable version of the MMDFII code. The first few months of MMDFII were quite rough and needed a great deal of "tender loving care".

For reasons that will be clear in a moment, this stable version of June 1983 is now referred to as the MMDFII-pre-England version. Around June, a copy of this stable version was delivered to Steve Kille of University College London (UCL) and Brendan Reilly of UDEL, who had taken over Dave Crocker's position regarding MMDF at UDEL. Steve Kille made a number of major changes to the handling of domains, address parsing, and handling of the alias files. Steve also added support for NIFTP, a European file transfer protocol used for sending mail in a batch environment. At the same time that Steve was making his enhancements, Doug Kingston continued to develop BRL's copy of MMDFII to make it an even more solid mail system. BRL's changes were not as major as Steve's but covered a great deal of code and fixed several major outstanding bugs. This dual development led to two variants of MMDFII that both needed the other's improvements. In late September of 1983 Brendan Reilly and Doug Kingston spent a week in England with Steve to merge the variants and to discuss future changes and directions for MMDF. The result of this meeting was a merged version of MMDFII which I will call MMDFII-post-England. Just prior to this trip, the CSNET Information Center (CIC) received a copy of the pre-England MMDF. Their later changes were based on this pre-England version which made merging of their changes into the post-England version somewhat difficult.

After the England meeting, Brendan Reilly of UDEL took the role of coordinator of the subsequent changes to MMDF. Copies of the MMDF-post-England were made simultaneously available to BRL, UCL, and UDEL. Since then many minor changes have been made by all four sites; in essentially all cases these changes have been bug fixes or changes to make MMDF a more stable and robust system.

In the past two months, Doug Kingston at BRL has made changes to the local delivery mechanism, rewriting much of the original code, and the central delivery program has been upgraded to take advantage of large address space machines when possible to keep retry histories for messages on a host-by-host basis. Bernie Cosell at the CIC has undertaken to speed up MMDF execution by providing a binary configuration file as an alternative to the ASCII text based version. Steve Kille has continued to refine the address handling and the British backwards domain code.¹ Brendan Reilly has made changes to the package to allow it to run on the Altos system and has fixed numerous bugs in the Phonet code. He has also been coordinating source code changes and merging the various changes to build a distributable system, which will be released when stable, quite likely before the release of this paper.

The MMDF Design

The MMDF system design has not changed fundamentally from the original design proposed and implemented for MMDFI. The design of MMDFI is covered in detail in the paper "An Internetwork Memo Distribution Capability".² In this chapter I will summarize the basic design and

¹ The British do domains backwards. For example, if in the US (Internet) we write "user@VAX1.EE.UDEL.ARPA" known as "little endian" order, the British (SERC Net) write "user@ARPA.UDEL.EE.VAX1" or "big endian" order.

² D. Crocker, E. Szurkowski, and D. Farber, "An Internetwork Memo Distribution Capability", Datacom Conference, September 1979.

discuss those changes that have been made in MMDFII.

Mail software is normally classed into one of two groups. A user agent (UA) is a program that is responsible for providing a "user friendly" interface for reading or writing mail and converting to the canonical interface of the mail transfer agent as necessary. A mail transfer agent (MTA) is a program or system to which you or your user agent entrusts a message for delivery to someone else's user agent. There has been a great deal of confusion caused by people who fail to realize the difference between a mail transfer agent and a user agent, or even that a difference exists! There is a wide variety of user agents which can be used with MMDF and it is their responsibility to provide a user interface. This separation of the functions of user agent and mail transfer agent has many advantages, not least of which is that MMDF can support many different user interfaces with ease. Currently, there are at least five different interfaces available including the Rand MH system, V6 style mail, Berkeley's Mail (cap-mail), the Tenex style Send and Msg programs, and Rmail (for UUCP). MMDF is a mail transfer agent. MMDF does not have nor does it claim to have a good "user interface"; instead it has a good program (MTA-UA) interface. MMDF accepts messages for delivery either locally or to a remote site. It attempts to verify the validity of the addresses at submission time to the extent possible given only a host table and a list of local addresses. If accepted it will continually try to resend the message until the retry time is exhausted at which time the message is returned to the sender.

The MMDF system can be thought of as two subsystems responsible for mail submission and mail delivery respectively. Between these two halves is the mail queue. The mail queue will be discussed later, but basically it stores each message as two files, an address list with some control information, and a separate file containing only the message text (header and body).

The submission half of the system consists mainly of one program, called "**submit**", which is responsible for enqueueing mail to be delivered. As much verification as possible is performed on the message at submission time. For mail destined for the local machine, this means making sure the destination account exists, and that any local mailing list or aliases expand properly. For mail that has a non-local host specification, the **submit** process checks to see if it knows how to reach the specified host. Mail which for any reason is known to be undeliverable, is not accepted for delivery. **Submit** is called by two types of processes. The first group includes user agents such as the **Send** program. The second group comprises channel programs such as **rmail**³ which are interfacing to remote mail transfer agents.

The delivery portion of MMDF is represented by two main elements: the **deliver** program which manages the queue, and the channel programs which handle the details of delivery to a specific network, host, or mail system. The **deliver** program takes each message which is eligible to be delivered, and opens the appropriate address list. For each address in the list, **deliver** ensures that it is running the appropriate channel program and then passes the envelope information⁴ to the channel. The message text is passed as a reference to the actual file which the channel program has access to. The channel decides how to deliver the message and sees to any necessary message reformatting that may be necessary (a.k.a. "header munging").

There is currently a variety of channels and the number is growing. The local channel handles delivery of messages to local addresses. The list channel is a special, somewhat incestuous, channel which acts the role of channel, receiving user agent, and sending user agent all in one. The list channel resubmits mail back into the mail system by calling **submit**. This has several benefits that will be discussed later.⁵ The SMTP channel delivers mail via TCP/IP connections using the SMTP protocol.⁶ The phone channel uses the Phonenet packet protocol dialup software developed at the University of Delaware for sending mail over dialup or hard-wired

3 The **rmail** program (/bin/rmail) is invoked by uucp when delivering mail on your system.

4 The MMDF envelope information consists of the the return address, the destination addresses, some delivery options and a reference to the message text file.

5 See the section on the list channel.

6 J. Postel, "RFC821 - Simple Mail Transfer Protocol", Network Information Center, SRI International, August 1982.

terminal lines. The NIFTP channel queues mail as files to be transferred using the NIFTP protocol used in the British research community. The UUCP channel is used to queue requests for transfer using UUCP.

Development of MMDFII was started about the same time Sendmail was being written by Eric Allman at Berkeley. Dave Crocker met with Eric on a number of occasions and was impressed by his work. Some elements of the Berkeley software were so useful that they inspired the development of similar facilities in MMDFII. Not the least of these was the runtime configuration file. It is now possible to totally configure the MMDF software from a single, ASCII text based configuration file. Unlike Sendmail's terse configuration syntax, MMDF uses much more verbose keyword/value pairing for configuration information. MMDFII can be configured from either compiled in values for fast startup or totally from the text configuration file or any combination of the two. As a result, the common MMDF tailoring file is one tenth the size of a Sendmail tailoring file or can be even smaller even on a large relay site. The runtime configuration file is one of the most useful additions in MMDFII, especially for sites supporting more than one host since one can now run the same binaries on all machines of the same type. Another Berkeley-inspired facility was the ability to have an alias file entry that forces a delivery to a file or pipe. While initially insecure, this facility has been made reliable by adding code to submit that knows when you are processing an alias file entry and when you are processing some other type of address. Simple ownership of the file containing the address was not considered sufficient protection since there are too many files left writable to the world that are owned by root or other privileged users.

The MMDF File Hierarchy

The MMDFI queue structure consisted of three directories. The "msg" directory contained the files containing actual message text. The "addr" directory contained the address list file for each message, and the "tmp" directory contained the address list while it was being created before moving it to the "addr" directory. The names of the files in "msg" and "addr" were identical although the contents differ. This made it easy to find the companion file given either of them. The MMDFII queue structure has changed in one major way from that of MMDFI. There is now one extra directory per channel named *q.channel*. If an address list still has any addresses destined to be sent on any channels, then a link will exist from the address list file in "addr" to a file with the same name in each channel directory which is referenced. All the above directories nominally live in /usr/mmdf/lock/home, although the root name of this tree can be changed. The lock directory is kept in 700 or 770 mode, accessible only to the "mmdf" user and possibly a "systems" group for ease of maintenance. The home directory and all the underlying queue directories are kept in 777 mode to allow ease of movement and access for the trusted but unprivileged programs that operate there. In particular, the **submit** process is setuid to "mmdf" to allow it to enter the tree, but it then restores its UID and GID to those of the invoker. **Deliver** and the channel programs normally run as the "mmdf" user. Only the local channel, which must access the real user's mailbox files, and the TCP/IP network daemons, which must access privileged sockets, need run privileged. There are several other programs that are setuid "root", but only so they can change their UID to "mmdf" so as to be a "trusted submitter", which allows them to specify an arbitrary From: line.

The addition of the separate queuing directories on a per channel basis was a valuable change. UNIX performs poorly with large directories as any UUCP backbone site can tell you. This new mechanism allows easy repartitioning of channel activities into separate directories, since **deliver** will never access the copy of the address list in the "addr" directory until it goes to finally expunge the message from the queues. If one channel or site gets backed up it does not affect the performance of any of the other channels.

The format of the address lists has changed somewhat since MMDFI. The old format was:

S T channel-name host-on-channel address-at-host

where S was either a '-' indicating unsent or a '+' indicating that only the address but not the

text had been sent. The T was either the type of delivery (almost always 'm' for mail) or a '*' indicating the message has been completely delivered. The channel-name and host-on-channel should be obvious. The address-at-host parameter was the local part only of an address. The new version differs in two ways. First, the host-on-channel is now the full domain address of the host to deliver to as specified in the routing information of the domain table.⁷ Second, the address-at-host is now simply the full address with both the local-part and the domain specifier.

In the future, I will probably change the location of the "message completely delivered" flag to be in the first position (S), since I feel it was a mistake to not have it there in the past and it is confusing in its current position.

The MMDF file hierarchy also has a logging directory. Separate logs are kept here for **submit** and **deliver**, the channel programs, and the phone dialing package. This directory is generally accessible although unreadable and the logs are normally write only. This could be made "mmdf" access only for some sites, with the only penalty being that some programs would be unable to add entries to the log. A subdirectory of the log directory called "log/phase" contains time stamp files whose modification times are changed by **submit** and **deliver** to indicate such things as last pickup time, last delivery time, last poll made, and similar information.

There are two other directories in the MMDF hierarchy which should be mentioned. The "chans" directory contains all of the channel programs invoked by the **deliver** program, and other ancillary daemon programs such as the SMTP daemon and the SMTP server. The "table" directory contains all files necessary to maintain the MMDF database, including domain tables, host address files, mailbox alias files, dialing scripts, and programs to build these files and incorporate them into the DBM library.⁸

Use of some sort of keyed database system is almost essential for large MMDF systems, since the table lookup overhead is unbearable otherwise. Currently DBM is the only readily available alternative and it does seem to work well, but it is running out of steam particularly due to its limited record length. A more flexible replacement for this package would be welcome.

The top of the MMDF directory tree contains the directories mentioned above, the **submit** and **deliver** programs, and a few maintenance programs. If the site is polled for Phonenet mail then the "slave" program is normally also located here. The **slave** program is used as the remote site's login shell and acts much as **uucico** in managing the link level communications. It in turn calls upon **submit** and **deliver** to send and receive mail.

Submit

For all the changes to its internals, the external interface of **submit** has changed remarkably little since MMDFI. The only notable external changes to **submit** are that it now processes domain style address (both forward and reverse!) and both RFC822 and RFC733 format addresses.⁹ A couple of new options have been added to **submit** to give some control over the handling of returned mail in the event that a message cannot be delivered. This is useful if the user is a program and does not care if the message is undeliverable! It is also now possible to feed a bare message to **submit** and tell **submit** to find all the addresses and show them and their validity on a one-by-one basis. This makes it convenient to feed mail to **submit** from a smart mail composer that doesn't want to know how to parse addresses, (a major task these days!).

The **submit** program can operate in one of two modes. In *protocol* mode, **submit** accepts options, a return address, and optionally a list of addresses for each message, followed by the message text. Multiple messages can be submitted one after another without reinvoking the **submit** process. Each address is individually acknowledged. If there is an error, the submission of that

⁷ See the section on the MMDF database for more information.

⁸ The DBM package is a set of simple hashed database access routines that were distributed with V7 Unix and are still widely used.

⁹ D. Crocker, J. Vittal, K. Pogran, D. Henderson, "RFC733 - Standard for the Format of ARPA Network Text Message", Network Information Center, SRI International, November 1977.

letter is aborted and a new submission may be made. In *one-shot* mode a single message is submitted on the standard input. As in *protocol* mode, it can be preceded by options and addresses, or the options can be given on the command line and the addresses taken from the message text.

The internal address verification process of **submit** has changed greatly since MMDFI. Most of the changes have been made to properly support domain style addresses. Additional changes were made to support per-channel and per-user access controls. While **submit** is checking each address regardless of origin, it is also compiling the address list for the message. Each address list entry contains the destination domain,¹⁰ the destination mailbox, and the channel in whose channel table **submit** first found the destination host. The lookup is somewhat complicated. The destination domain is looked up in the domain tables; the first entry found is used. Each domain table entry has associated with it the routing to be used to reach that domain/host. Normally this is just the name of the host itself if it is directly accessible, but it can be a sequence of hosts if the destination is not directly accessible. This "routing host" is then looked up in the channel tables to find a channel which can reach it. The routing host specifications and the entries in the channel tables are always full domain specifications so as to be unambiguous.

Authorisation is checked after a valid channel for a host is found. Access to send via a given channel depends on the originating channel and/or the submitting user. If access to a given channel is denied, **submit** will continue to look at subsequent channels to see if some other channel has access to the same host and is authorised. This mechanism is commonly used to restrict access to expensive transport systems or to restrict message transfer between channels representing private and public data networks.

Deliver

The **deliver** process has changed little until the past two months, when a dead-host caching facility and advanced retry mechanisms were added to **deliver**. Until then, the major changes to **deliver** were bug fixes and changes to enhance error recovery. Most notably, the mechanism to return mail was made much more persistent. In MMDFI, if a message could not be returned to the sender, it was "dropped on the floor". This is not acceptable. The new mechanism first tries to return to the sender; if that fails, an attempt is made to send the message to the local "Postmaster" address. If this too fails, then **deliver** tries to write the message into a "dead letter" file (usually `/usr/mmdf/lock/home/DeadLetters`). The last resort is to scream loudly into the log files.

The recent changes to **deliver** are designed to reduce **deliver**'s load on a system that handles a large amount of mail. The first change is to move the dead-host caching function out of the channels (currently only SMTP) and into **deliver**. This has two advantages. First, **deliver** no longer has to hand every address to the channel to find out that the host is dead. This was expensive since communications between **deliver** and **submit** are interactive using pipes and thus involved a great deal of context-switching and pipe I/O. Second, the dead-host caching is now a generally available facility that can be used by any channel without duplication of code.

The second change is to add a mechanism for storing the retry history for each dead host on each channel including retry count and last retry time. From this information it is possible to implement intelligent retry strategies using exponential backoff and maximal retry times. This greatly reduces the overhead caused by recalcitrant hosts that, are unavailable over a long period of time. The retry history change is conditionally compiled into **deliver** since it can by design use quite a bit of memory if there are a lot of dead hosts or pending messages (or both!). Machines with a relatively limited address space may not be able to use this feature.

10 By "domain" we mean the full domain specification of a host, e.g. VAX1.UDEL.ARPA.

The Local Channel

The local channel had remained unchanged until recently when a major reworking of the channel was done by BRL. The old local channel could handle delivery in three basic ways. The first and most common was to deliver directly to the user's default mailbox, appending the message to the end. Second, the user could put a program in his private bin directory called "rcvmail" that would be called by the local channel to handle delivery of the message. If the user program (rcvmail) did not complete successfully, a standard delivery was made instead.

In early versions of the local channel for MMDFI, Dave Crocker added support for mailing to files and pipes, but the original version had a number of security problems, mostly due to **submit**, so the capability was not much used.

The latest version of the local channel has kept the alias file-originated delivery to files and pipes, and the changes Steve Kille has made to the **submit** program have also made this facility reliable from a security standpoint. The rcvmail mechanism has been totally scrapped in favor of a more general and powerful mechanism which I will call "mail delivery files".

Mail delivery files were designed to give the user as much flexibility over how his mail was delivered as possible without opening security holes. The mail system was changed to allow local addresses to have a suffix appended which consists of an '=' and any simple text; this suffix is totally ignored except when using mail delivery files. Each user may create a ".mailedelivery" file in his home directory which contains one or more delivery specifications. A delivery specification is comprised of four parts as shown here:

address *action* *A/R* "*optional string*"

The *address* is used to match against the address being delivered to *including* the suffix (e.g. "dpk=unixwizards"). If the user subscribes to different lists with different suffixes he can use his mail delivery file to segregate his mail by source. To do this based on the message text alone is impossible to do right 100% of the time. The *action* is currently either "file", "pipe", or "destroy". The "file" action writes the message in standard mailbox format into the file specified in the optional string. The "pipe" action causes the program in the optional string to be run with the message available on the standard input. The "destroy" action causes the mail to be thrown away silently. This is useful if you go away on a long trip and don't want to unsubscribe to lists, but also don't want to come home to several thousand messages. The *A/R* flag (one character, either 'A' for accept or 'R' for reject), indicates whether the action, if successful, is sufficient to mark the message as delivered. If the local channel has tried sending the message to all matching delivery lines and the end of the .mailedelivery file is reached with the message still undelivered, then a standard delivery is made to the default mailbox. This protects against mail being lost due to users making errors in their .mailedelivery files.

The local channel will try to send the message to all delivery lines for which the address portion matches the destination of the incoming message. This means that a user could have multiple actions take place for any given message. For example, the user could have a TTY alert message sent to his terminal and also have the message resent to his new home machine by the following .mailedelivery file:

```
dpk pipe R "/usr/mmdf/mailutils/ttyalert"
dpk pipe A "/usr/brl/bin/resent dpk@brl-vgr.arpa"
```

The last line, if completed without error (a return code of 0 from resend), would mark the message as delivered due to the A (accept) flag in the third column.

The List Channel

The list channel was developed as the result of BRL's experience in managing large Internet mailing lists. Two major problems were discovered with dealing with mailing lists in MMDFI. First, since **submit** always verifies all known addresses for a message at submission time, if you were on the machine with a large list and submitted mail to the list you would have to wait for every address on that list to be verified. On a busy machine with a list of hundreds of addresses,

this could take five or ten minutes. Annoying as this may be for people, the situation was worse for mail submitted over communications channels like TCP/IP. The remote end would continually time out before the message had been completely verified so the message could never be sent.

The second problem with large lists was that there were always rejected mail notices going back to those least able to do anything about the mail problem, the original sender of the message. What was really desired was a method to try to have returned mail go to the list maintainer instead of the message's original sender.

The solution to both of the problems is embodied in the list channel. This is a channel with an incestuous relationship to **submit**, **deliver**, and the alias file. Use of the list channel is best described in parallel with the special entries for the list channel in the alias file. If we were maintaining a large list called "biglist", the following entries would be in the alias file:

biglist:	biglist-outbound@list-processor
biglist-outbound:	</usr/mmdf/lists/biglist-file
biglist-request:	maintainer

The pseudo-host "list-processor" has its own domain table and its one host table but represents no actual host. If someone submits mail to biglist, **submit** will find the alias entry and upon finding that it's not a local address, will queue it to the host "list-processor", so verification is complete after only one lookup. Unknown to **deliver**, the list channel simply calls **submit** and feeds it the aliased addresses, "biglist-outbound". This time the actual verification is done on the contents of the address list "biglist-file". Since the list channel is processed by a background daemon, no one is forced to wait through the verification process except the background daemon itself, which doesn't care how long it takes so long as it completes.

The list channel also performs another function to try to eliminate the problem of failed mail messages. For each address given to it, (normally just one), the list channel sees if there is matching "listname-request" entry in the alias table. It knows enough to try stripping any "-outbound"s from the name first though. If a "-request" entry is found, then that address is substituted instead of the original return address. The message text is *not* altered, but the new return address is recorded for use when resending the message. The new return address is supplied in SMTP "MAIL FROM:<address>" commands and any other situations where the return address is directly specifiable.

The changing of the return address is only useful when mail is rejected when submitted to the foreign host or if that host is smart enough to keep the return address information around. Many hosts do not maintain this information, and many of the same hosts are also promiscuous in that they will completely accept a message containing total garbage and decide to tell you about it later. This is precisely what MMDF tries to avoid by submission-time verification.

The UUCP Channel

The task of integrating UUCP mail into MMDF was a prime goal for BRL. Our users would not tolerate having to use two radically different mail interfaces for two different kinds of mail connections. We decided to write a channel to interface to the UUCP mail world that would take care of the necessary format conversions to allow mail to traverse the two mail worlds. The channel has two parts. The input portion of the channel is the program **/bin/rmail** which is executed by the UUCP program **uuxqt** when mail is being delivered. The output portion is a standard channel that invokes the UUCP system after reformatting the message.

The **rmail** program has been totally rewritten to interface to MMDF. **Rmail's** greatest task is to collect and reformat the address strings in the message. To reformat each address, **rmail** uses the UUCP channel table to determine what hosts are known to this host and shortens an host!host!host! string down to the single most distant host we know about and any subsequent hosts we do not know. For example knownA!knownB!knownC!unknown1!unknown2!user would become knownC!unknown1!unknown2!user. It then converts this to a RFC822 style address by putting the unknown hosts and the user in the local part and putting the known host with a domain in the domain portion, thusly: unknown1!unknown2!user@knownC.UUCP. If all the hosts

are known, then only the user is left in the local part, and the address winds up being user@known.UUCP. Since you always know the hosts you talk to, you can build any arbitrary UUCP path by simply saying arbitrary-host-path@neighbor.UUCP.

Rmail is prepared to accept destination addresses in two forms. If the addressee is just another UUCP host addressed using host!host!... notation, then **rmail** forwards the letter via UUCP without header munging since the destination host may not support RFC822 style mail. An addressee of the form user@domain will cause the message to be fed to **submit** and into MMDF proper where the message can be delivered to another UUCP site or any other site MMDF can get to that is not restricted by the **submit** authorisation system. **Rmail** will reformat the message header in the latter case to conform as much as possible with the RFC822 specifications.

The outbound portion of the UUCP channel is a mmdf channel program called "uucp" which is invoked by **deliver**. The job of this program is much easier since all it must do is reformat the from line to be compatible with UUCP mail. The outbound channel must also reformat the destination addresses which become arguments to **uux**. The outbound channel uses the same channel table that **rmail** used but performs the reverse action on the address so, for example, root@mcnc.UUCP becomes uncl!duke!mcnc!root and this is then further divided to form the **uux** command "uux uncl!rmail duke!mcnc!root" (assuming the channel table maps mcnc.UUCP into duke!uncl!mcnc).

The UUCP channel would have to be classed as the only "flakey" portion since some of these address transformation really need an advanced AI system to make an intelligent transformation, but in general the channel does a very good job and has little trouble with "normal" UUCP addresses.

Using Domain Name Servers

The use of domain name servers will have some interesting effects on the address verification aspects of mail submission. In the current system, all the information necessary for verification of address is in a local data base. When we are using name servers, we can no longer be guaranteed that all the needed information will be locally cached. In addition, we are not guaranteed that we will be able to reach all the necessary name servers at submission time (although duplicate name servers will make this possibility small). The **submit** program will call the local domain resolver to verify each address, and there will be some time limit in which to complete this task. The resolver will be expected to first consult the local cache of domain data and if the information is not found, contact as many servers as necessary to resolve the address.

The possible lack of information will force us to provide a contingent submission queue for those messages that are not able to be verified at submission time. This does not imply that there will be no verification. We will verify that we at least know the top level domain of each address and verify each sub-domain when possible. If some sub-domain of the full address is known to be bogus, the address can be flushed. Knowing that we have authoritative information that a domain does not exist is just as important as knowing that it does exist.

A new channel much like the list channel will be used to process the partially accepted address for a message. This channel will continually try and verify the address until it is known good or bad and will have the message returned to the sender with an explanation if one or more addresses is bad. Most systems will run with a fairly rich cache of host information. For those systems which cannot afford to keep this information around, the submission time verification might be a considerable delay which would be unacceptable for a user interface. On these systems it will be possible to always queue the message for background verification (via the "verification" channel).

Conclusion

MMDFII had some early problems and as a result may have gotten some initial bad press, but MMDFII has shown that it is a capable mail system which is both robust and able to handle very large mail loads. There now exist a growing number of tools to analyze and manage large

flows of mail in a MMDF system. These tools include status programs, sophisticated logging, and log analysis programs. Because of the separation of mail into separate queues, multiprocessing of the mail queues is not only possible, but routinely used to both increase throughput and decrease delays. MMDF is also a flexible system. Runtime reconfiguration is simple, generally easy to understand, and can be done at any time. Since the MMDF core software is free of channel specific or network specific information, one can easily add additional channels for new networks or protocols without effecting the existing software. MMDFII represents a stable, production mail system, providing a strong base for the development of new network interconnections and mail handling environments which are essential in today distributed computing environment.

The MMDFII software is available under license, free of charge, (with the possible exception of a tape copy fee), for internal use only as follows: to U.S. Government agencies through the Ballistic Research Labs, to CSNET sites through the CSNET Information Center at BBN, and to others through the Dr. David Farber at the University of Delaware, Electrical Engineering and Computer Science Department. Commercial concerns interested in MMDF for other than internal use should contact Dr. Farber.

DRAGONMAIL: A Prototype Conversation-Based Mail System

Douglas E. Comer and Larry L. Peterson
Purdue University

ABSTRACT

We describe a prototype user interface to computer mail based on conversations among individuals. Conversations provide a higher level organization to messages than conventional memo-based mail systems. Messages in a memo-based system are treated independently, and presented to the user ordered by their arrival at the user's mail-box. Any relationship that may exist between memos must be explicitly observed by the user. In contrast, a conversation-based system organizes mail before the user sees it by grouping messages into conversations. In addition, messages within a conversation are ordered based on the *context* in which they were written.

1. Introduction

Computer mail — the exchange of messages among computer users — has gained acceptance as a viable medium for communication among individuals throughout the world. It offers the advantages of written communication at electronic speeds, while remaining less expensive than telephone communication.

Conventional computer mail can be conceptually viewed in the following manner. A user, called a *sender*, first composes a message with the assistance of a *User Interface (UI)*. The user interface then submits the message to the *Message Transport System (MTS)* for delivery to a *recipient* user. The *MTS*, which may span one or more machines, deposits the message in the recipient's *mailbox* where it remains until accessed. The recipient retrieves the message assisted by a user interface. (The user interface used by the sender and by the recipient need not be the same.) The sender identifies the recipient by specifying a *mailbox address*; the sender encloses a return address with the message, enabling the recipient to reply [DENN81, HEND79, LEVI79, PICK79, SCHK81].

Although the *MTS* can abstractly be viewed as a single communication channel, it actually consists of a confederation of interconnected delivery systems. Users on a single computer system share a *local* mail delivery system. If computer systems are connected by network hardware, they may cooperate to form a *network-wide* delivery system. Because computer systems are often connected to more than one network, network delivery systems overlap to form a *global* mail system.

To deliver messages through the *MTS*, each component computer system implements a hierarchy of *mailers*. Initially, a *global-mailer* [ALLM83, CROC79, POST79] accepts arriving messages. If the recipient is known on the local system, the global-mailer passes the message to the *local-mailer* [SHOE79] for final delivery to a mailbox. Otherwise, the global-mailer selects a suitable *network-mailer* [POST82, SCHM79, BIRR82] to deliver the message

This project is supported in part by grants from the National Science Foundation (MCS-8219178), SUN Microsystems Incorporated, and Digital Equipment Corporation.

to a remote system. The network-mailer forwards the message over the network to a remote system where the message is deposited with its global-mailer and the process is repeated.

2. Conventional User Interfaces

The transport system just described has the potential to support a large flow of information among individuals. It is the purpose of the user interface to assist the individual in the management of this information by providing various mechanisms for viewing received messages, composing messages to be sent to others, and archiving old messages for future reference.

Many user interfaces for computer mail currently exist [SHOE79, BORD79, CROC77, BRO781, ALME83, TYMS83]. They can be operationally defined in the following way. The local-mailer appends new messages to the user's mailbox. When executed, the user interface retrieves the individual messages and displays a brief summary of each. The user may then display the contents of selected messages. After viewing a given message, the user can either save or delete the message, as well as generate a reply. In addition, the user interacts with the interface to create new messages.

User interfaces differ in the way they archive messages. Simple interfaces, such as UNIX mail, place saved messages in a named file. Other interfaces like MH, MS, and Laurel provide more advanced mechanisms for archiving messages that the user wishes to save. Users define categories of messages, usually called *folders*, into which messages are filed. Further commands allow the user to browse through this hierarchy of messages, and to search for particular messages based on a specified attribute-value pair. In addition, user interfaces like EDMAS and AUGMENT provide a second mechanism for accessing related messages by *linking* together messages designated as replies to previous messages. The user is then able to traverse through the list of related messages.

Despite their differences, conventional user interfaces are all founded on a *memo-based* model, similar to the conventional office practice of memo communication. Each memo that arrives for an individual is treated independently from all other memos. Furthermore, memos are displayed in the order in which they arrive. The user must observe any relationships that may exist between memos; the user interface only provides assistance in managing memos *after* they have been read.

In addition to the conventional memo-based model for mail, systems like FORUM and Network-News [LIP74, HORT79] support the exchange of information among users based on a *teleconferencing* model. Rather than sending a memo to a set of recipients, users *contribute* messages to *dissussions* of which they are a *member*. All members of a discussion are then able to view the messages contributed to the discussion. A limited teleconferencing service, called a Bulletin-Board, provides a single discussion group for all users on a computer system. Users are able to post notices on the bulletin board which all other users on the system are then able to read.

The underlying structure that supports the memo-based and the teleconferencing-based system is the mailbox and the discussion, respectively. In both cases, the interface treats messages in a First-In-First-Out manner by appending new messages to the end of the structure and removing messages for viewing from the beginning. Thus, the order in which users see messages is dependent on the order in which they arrive at the mailbox or discussion.

3. Objectives

DRAGONMAIL is a prototype user interface that replaces both the memo-based model and the teleconferencing-based model with a conversation-based approach. Conversations provide a more powerful data structure that supports a higher-level organization of messages. Before describing the features of *DRAGONMAIL*, we present a set of goals we believe a user interface should meet.

- Messages should be grouped together into conversations when presented to the user. Conversations are more consistent with the way humans communicate.
- All forms of mail-like services such, as teleconferencing systems, bulletin boards and news, should be incorporated into a single, uniform system.
- The order in which groups of related messages are presented to the user should correspond to the importance of the messages. The user should be able to determine what mail is urgent so he can process it first.
- Mail should be archived so that it can be easily referenced. The user should be able to query the mail archive for all messages discussing a particular subject, from a certain person, etc.
- The history of a conversation should be maintained to allow new users to be included in a discussion.
- The context in which a message was written should be available. That is, there should be a mechanism for determining what messages the sender viewed before composing a given message. Furthermore, the user interface should use the message context to present messages in a meaningful order.
- Messages and conversations should be displayed in a clean and easily-readable format. Multiple windows should be used so that relevant information can be easily extracted, and messages should not be cluttered with unnecessary header information.
- Irrelevant and unnecessary messages should be eliminated so as to reduce the volume of mail with which the user must deal. For example, messages should have expiration dates so that out-of-date notices are automatically deleted. Also, it should be possible to replace a group of messages with a single summarizing message.
- The user interface should display names in a form acceptable to the user so that the participants in a conversation are easily identifiable.

4. Attributes of DRAGONMAIL

We now describe a prototype user interface called *DRAGONMAIL* which places a high-level structure on computer mail. The fundamental object of communication is the *conversation* rather than the memo. Instead of reading, writing, and filing individual memos, users participate in a set of conversations.

A conversation consists of a group of *messages*, denoted $M = \{m_i \mid i \geq 0\}$, which are shared by a set of *participants*, denoted $P = \{p_i \mid i \geq 0\}$. Participants are able to view all messages associated with the conversation, and as well as add new messages. Thus, we take the view that messages are *submitted* to a specific conversation rather than mailed to a group of recipients.

Each conversation has a *topic* and each message has a *subject*. Whether a conversation is actually restricted to a single topic or not is a decision made by the participants. It is envisioned, however, that this will be the case as users are free to have more than one conversation with the same participants at the same time.

A conversation begins when a user defines the set of participants P and submits an initial message m_0 . New members may be added to a conversation by having that list expanded to include them. Similarly, old members may be removed. Being added to a conversation means having access to the entire *history* of the conversation, (i.e. all of set M). Removal implies not being able to read any future messages submitted to the conversation.

Each participant in a conversation is known by a *universal-identifier* which is understood by the underlying mechanisms that support *DRAGONMAIL*. In addition to universal ids, a private set of *aliases* is maintained for each user. These aliases are used by a participant to specify other members of a conversation, and by the user interface when presenting participant names to the user.

At any time, a given user will be participating in a set of conversations. In contrast to a memo-based system where messages are perceived to be either "new" or "old", the set of conversations is partitioned into *foreground* and *background* subsets. A conversation is said to be in the foreground if a participant has acted on it in the past n days, where n is a parameter of the system. Foreground conversations are further partitioned into those that contain messages not yet seen by the user, and those in which the user is up-to-date. Background conversations are those that have been idle for n days.

The set of messages which make up a conversation are classified as being either *visible* or *hidden*; denoted M_v and M_h respectively. Visible messages are automatically displayed and made available to participants when a conversation is viewed. Participants are able to hide certain messages when they are determined to be irrelevant to the conversation. Hidden messages are still maintained in the history of the conversation, and may be examined with special commands, but are not automatically displayed to the user.

The participants in a conversation are classified according to the privileges they have. By default, all participants are granted the right to read the component messages of a conversation. Such participants are said to be *passive*. If the participant also has the right to submit new messages into a conversation, the participant is considered *active*. Certain participants, called *administrative* participants, are granted the right to affect the conversation in ways other than by submitting new messages. For example, they may add new participants and move irrelevant messages into the set of hidden messages. Finally, the participant that initiates a conversation is called the *owner* of the conversation. Besides having administrative authority, the owner has the right to grant the privileges mentioned above to other participants.

The underlying structure of *DRAGONMAIL* maintains the relationship among the messages which make up a particular conversation. Informally, when a participant composes a message, he does so in the *context* of the messages he has already seen. Specifically, *message context* is a relation that holds between messages i and j , if and only if m_i had been read by the author of m_j before composing m_j . The set of such relations for all the messages in a conversation can be represented by a directed acyclic graph called a *context graph*, where the vertices correspond to the messages and the edges of the graph represent the message context relation [COME84]. The context graph is then used to present the messages in a given conversation to the user in a meaningful order.

It should be noted that the degenerate case of *DRAGONMAIL* is the same as memo-based mail. For example, a user can start up a new conversation by simply sending an initial message. If replies are also sent in separate conversations, a set of independent conversations results. Such a set of conversations is equivalent to the set of independent memos generated by current memo-based user interfaces. Furthermore, basing mail on conversations leads to a scheme for incorporating all forms of mail-like activities into one format. For example, current system-wide bulletin boards can be replaced by a single conversation, with all users on the system as participants. Similarly, each news group that a user belongs to can be managed as a conversation.

5. Operational Description

This section presents a high-level description of the set of operations available in *DRAGONMAIL* for managing conversations and messages. We classify them into the three *levels* of operation viewed by the user.

5.1. Operations on Sets of Conversations

The first group of commands are used to manage a set of conversations belonging to a given user.

- **LIST-CONV**(*conversation-set*)
- **SELECT-CONV**(*conversation*)
- **MERGE-CONV**(*conversation*₁, ..., *conversation* _{n} , *topic*)
- **SPLIT-CONV**(*conversation*, *topic*₁, *topic*₂)
- **CREATE-CONV**(*topic*, P , m_0)
- **DELETE-CONV**(*conversation*)
- **QUERY**(*attribute*, *value*)

Upon entry into *DRAGONMAIL*, the screen illustrated in Figure 5.1 is displayed. It contains a brief synopsis of the conversations in the foreground. Lines are highlighted, (by boldface in this paper,) to indicate that the conversation contains an unread message. Conversations marked with an asterisk contain urgent messages.

CONVERSATION	TOPIC	UNREAD/TOTAL
1	TILDE project reports	2/15
*2	Postmaster Duties	10/123
3	Header-People	99/23014
4	Ethernet on 8086s	0/18
	.	
	.	
	.	
Command Line Window		

Figure 5.1
Initial screen displayed by *DRAGONMAIL*

The user selects one of the conversations for further consideration. Selecting a conversation causes the conversation level described in section 5.2 to be entered. Also, new conversations can be created and old ones deleted. Additional commands allow the user to control the direction of conversations. A conversation that has diverged into a number of subtopics may be divided, and two or more overlapping conversations can be merged together. Splitting a conversation means that two new identical conversations are created to replace the original conversation. Merging a set of conversations implies that the new conversation consists of the union of the messages in the original conversations.

Besides viewing conversations in the foreground, an explicit mechanism is provided for generating a subset of conversations. All of a user's conversations are stored in a database. The user may query this database for a set of conversations by specifying an attribute-value pair, where appropriate attributes are conversation *topic* and *participant*. One of the conversations produced by the query, and in general, a single conversation from any set of conversations, may be selected for further consideration by the user. The database can also be queried using attributes that cut across conversation boundaries, such as message *subject*, *date*, and *content*. In this case, a set of messages is produced rather than a set of conversations.

5.2. Operations on a Single Conversation

The following group of commands are used to manipulate a single conversation.

- **LIST-MSGs**(*message-set*)
- **LIST-PARTS**
- **SELECT-MSG**(m_i)
- **CONTEXT-MSG**(m_i)
- **SUBMIT-MSG**(*subject*, m_{new})
- **WITHDRAW-MSG**(m_i);
- **ADD-PART**(p_i , *permission-class*)
- **REMOVE-PART**(p_i)
- **CHANGE-PERMISSION**(p_i , *permission-class*)
- **SUPERSEDE-MSGs**(m_1, \dots, m_n, m_{new})

The screen in Figure 5.2 is displayed when a specific conversation is selected for detailed viewing. It contains a brief synopsis of messages in the conversation, where unread messages are highlighted, and urgent ones are marked with an asterisk. *DRAGONMAIL* performs a topological sort [KNUT73] on the context graph for the conversation to produce the ordering of messages. Previously read messages (i.e. non-highlighted) appear in this display to give the surrounding context of new messages. The user can both select specific messages for further consideration and submit new messages to the conversation.

Conversation 2: Postmaster Duties	
MESSAGE	SYNOPSIS
1	Chris / March 17th, 9:30 / UNIX style from lines
2	Bonnie / March 16th, 18:44 / Mailing list request
*3	Doug / March 17th, 18:50 / BITNET Relay
4	Tony / March 17th, 15:31 / ECN domain
	.
	.
Command Line Window	

Figure 5.2
Screen displaying a specific conversation

The **SUPERSEDE** operation gives participants the ability to replace a group of messages in a conversation with a single message. Such a mechanism allows a participant to summarize a discussion, and exchange that summary for the individual messages which led to it. The superseded messages still exist, but are *hidden* and not automatically displayed with the conversation. This command allows participants in a conversation control over the volume of irrelevant information that inevitably accumulates in computer mail.

5.3 Operations on a Single Message

Finally, we discuss the operations that can be performed on individual messages. Once a conversation and message have been selected using the operations defined above, the user can display the contents of that message and form replies. The following is a list of commands that can be performed on a single message.

- **COMPOSE**
- **ATTACH**(*file*)
- **SET-PARAM**(*parameter, value*)
- **DISPLAY**
- **DETACH**(*file*)

Unlike conventional memo-based systems which display a large number of header lines, (even some dealing with message transport), *DRAGONMAIL* mail displays only relevant information such as the message author, subject, and date, in addition to the message body, as illustrated in Figure 5.3.

Conversation 2: Postmaster Duties
Message 1: Chris / March 17th, 9:30 / UNIX Style From Lines
<p>Could you please take a look at how sendmail generates the UNIX Style from line; it doesn't work right when the message passes through a mailing list exploder.</p> <p>Cheers, Chris</p>
Command Line Window

Figure 5.3
Screen displaying a specific message

When submitting a message to a conversation, the participant generates the message by first composing the message body. The user may then set a number of parameters associated with that message. For example, a *time-fuse* parameter may be set, which causes the message to be removed from the conversation at a specified time, keeping out-of-date messages from cluttering conversations.

The composer can also assign a high *priority* to the message so it will be marked as urgent when listed by the other conversation members. Finally, a *reply* parameter can be set to denote the message as a reply to a specific message in the conversation. In addition to setting parameters, the user can attach files of information to a message, (e.g. programs, data, text-processing sources, etc.). Other members of the conversation then detach the files for further processing and execution.

6. Summary

We have presented an overview of a prototype user interface for computer mail based on conversations rather than independent memos. *DRAGONMAIL* groups related messages into conversations and organizes them according to their context. Conversations have the advantage of being more consistent with the way humans communicate. They also provide an overall structure to mail that allows all forms of computer-based communication to be presented in one uniform environment.

7. References

- ALLM83 Eric Allman, "SENDMAIL — An Internetwork Mail Router", *UNIX Programmer's Manual*, 4.2 Berkeley Software Distribution, Vol. 2, 1983.
- ALME83 Guy Almes, Andrew Black, Carl Bunje, and Douglas Wiebe, "Edmas: A Locally Distributed Mail System", Technical Report 83-07-01, University of Washington, July 1983.
- BIRR82 Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder, "Grapevine: An Exercise in Distributed Computing", *Communications of the ACM*, Vol. 25, No. 4, April, 1982, pp. 260-273.
- BORD79 Bruce S. Borden, R. Stockton Gaines, and Norman Z. Shapiro, *The MH Message Handling System: User's Manual*, R-2367-AF, Rand Corp., November, 1979.
- BROT81 Douglas K. Brotz, *Laurel Manual*, XEROX, CSL-81-6, Palo Alto, California, May, 1981.
- COME84 Douglas Comer and Larry Peterson *Conversation-Based Mail*, Tilde Report CSD-TR-465, Purdue University, March, 1984.
- CROC77 David H. Crocker, *Framework and Functions of the "MS" Personal Message System*, R-2134-ARPA, Rand Corp., December, 1977.
- CROC79 David H. Crocker, Edward S. Szurkowski, and David J. Farber, "An Internetwork Memo Distribution Capability — MMDF", *Proceedings of the Sixth Data Communication Symposium*, November, 1979, pp. 18-25.
- DENN81 Peter Denning and Douglas Comer, *The CSNET User Environment*, CSD-TR-456 Purdue University, July 1981.
- HEND79 D. Austin Henderson and Theodore H. Myers, "Issues in Message Technology", *Proceedings of the Fifth Data Communication Symposium*, November, 1978, pp. 6.1-6.9.
- HORT79 Mark Horton, "How to Read Network News", *UNIX Programmer's Manual*, 4.1 Berkeley Software Distribution, Vol. 2, 1979.
- KNUT73 Donald Knuth, *The Art of Computer Programming*, Vol. 3, Addison-Wesley, Reading Mass., 1973.

- LEVI79 Roy Levin and Michael Schroeder, *Transport of Electronic Messages Through a Network*, XEROX, April, 1979.
- LIP174 H. M. Lipinski and R. H. Miller, "FORUM: A Computer-Assisted Communications Medium," *Proceedings of the 2nd International Conference on Computer Communications*, pp. 143-147, August 1974.
- PICK79 John R. Pickens, "Functional Distributed Computer Based Messaging Systems", *Proceedings of the Sixth Data Communication Symposium*, November, 1979, pp. 8-17.
- POST79 Jonathan B. Postel "An Internetwork Message Structure" *Proceedings of the Sixth Data Communication Symposium* November, 1979, pp. 1-7.
- POST82 Jonathan B. Postel "Simple Mail Transfer Protocol", *RFC 821*, August, 1982.
- SCHK81 Peter Schicker, "The Computer Based Mail Environment—An Overview", *Computer Networks*, Vol 5, 1981, pp. 435-443.
- SCHM79 Eric Schmidt, "An Introduction to the Berkeley Network", *UNIX Programmer's Manual*, 4.1 Berkeley Software Distribution, Vol. 2, 1979.
- SHOE79 Kurt Shoens, "Mail Reference Manual", *UNIX Programmer's Manual*, 4.1 Berkeley Software Distribution, Vol. 2, 1979.
- TYMS83 Tymshare, Inc., Office Automation Division, "The AUGMENT Mail User's Guide", Journal Document <AUGMENT,103481>, November 1983.

Converting the BBN TCP/IP to 4.2BSD

Robert Walsh and Robert Gurwitz

BBN Laboratories

10 Moulton Street

Cambridge, MA 02238

ABSTRACT

BBN was contracted by DARPA in September 1980 to develop an implementation of the DoD network protocols TCP and IP for Berkeley UNIX on the VAX. The first versions of this software became available in May 1981. In Fall 1981, a version for 4.1BSD was turned over to Berkeley for integration with the new IPC and network subsystems being written for 4.2BSD. From that point on, the two versions diverged with Berkeley concentrating on performance in LANs and BBN focusing on optimal operation in the DARPA Internet.

Recently, we attempted to reconcile the two versions in 4.2BSD. The result is a 4.2BSD version that uses the BBN TCP/IP transmission and routing algorithms along with the Berkeley IPC interface and device driver structure in a manner that is transparent to the user and programmer.

This paper describes the experiences we had in converting the BBN TCP/IP to 4.2BSD. It compares the features of the two implementations and reports on performance results obtained from the new version in both LAN and long-haul environments.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Introduction

In September 1980, Bolt, Beranek, and Newman (BBN) began a project supported by the Defense Advanced Research Projects Agency (DARPA) to implement the DoD standard network protocols Transmission Control Protocol (TCP) [1] and Internet Protocol (IP) [2] for Berkeley UNIX[†] on the VAX^{††}. In performing this work, BBN has tried to provide a high performance, general purpose, complete implementation of TCP/IP. BBN endeavored to design this software as a modular extension that drops right into the UNIX kernel without affecting the rest of the kernel very much. We believe the current code meets these goals.

BBN's TCP is implemented by an explicit finite state machine. When something happens that affects a connection, the event and the connection's current state are used to index a jump table and to call the subroutine which deals with that specific state transition. This design makes the code quite supportable. Once a problem with the implementation has been well defined, a programmer can walk through the code and correct it.

The protocol is linked to the rest of the operating system by jump tables. User requests on internet sockets go through one table to get fielded by TCP/IP. Should the resulting state transition involve sending a packet, TCP goes through another jump table to call upon a device driver to send the packet. This modularity means that the implementation is isolated from the rest of the kernel, and that it is easy to substitute the BBN implementation for the Berkeley one.

Porting the BBN code to 4.2BSD

Getting to the point of easy substitution was not an afternoon's task, though. The data structures describing network interfaces, routes, sockets, and network buffering had changed in going from 4.1 to 4.2BSD. The system call interface also changed; e.g., *open* became *socket*, *bind*, *connect*, *accept*, and *listen*. In porting the BBN code to 4.2BSD, the goals were: to use the new Berkeley system calls and driver interface to make the substitution transparent; to preserve Berkeley's performance gains in Local Area Network (LAN) environments; to preserve the BBN TCP/IP's performance in long-haul environments; and to retain the modularity and supportability of the BBN code. We believe these goals have been met.

4.2BSD had been a dramatic re-write of UNIX. Modifying the BBN implementation so that it would work in the new environment, getting it to compile and pass through *lint*, took two months. Working out the bugs required two weeks to get TCP working and two weeks to get the code to interact well with other implementations.

Most of the difficult problems encountered involved *mbufs*. *Mbufs* are used in the kernel to buffer data for the user and to hold data structures allocated at run time. For BBN TCP/IP, *mbufs* are twice as large as for Berkeley TCP/IP so that TCP control blocks may fit inside a single *mbuf*. Checks are made at boot time to ensure that each network data structure fits inside one *mbuf*. In 4.1BSD, *mbufs* stored a small amount of information by pointing to a small buffer area within the *mbuf* structure itself. This internal buffer could store 240 bytes. With 4.2BSD, some *mbufs* can store a larger amount of data by pointing to a cluster of pages somewhere else in memory. If an *mbuf* points to a cluster of pages, it can store 1024 bytes. One resulting problem cropped up in some code which assumed that all *mbufs* can store the same amount of data.

[1] The Transmission Control Protocol provides a byte stream, a pipe in UNIX terminology, between two processes on two machines. Data is transported reliably, presented to the receiving process in the same order as it was sent, and checksummed to preserve data integrity. TCP uses the Internet Protocol to carry packets between the two machines.

[2] The Internet Protocol is a datagram protocol concerned with transporting bytes from one machine to another. Data may or may not arrive. Data may arrive in a different order from that in which it was sent. Data integrity is not guaranteed. IP splits a packet into fragments when the packet is larger than the maximum size supported by the local network on which it will be sent. Message fragments may themselves be divided if the message crosses a gateway to another network which has a smaller maximum packet size. At the destination host, message fragments are reassembled.

[†] UNIX is a trademark of Bell Laboratories.

^{††} VAX is a trademark of Digital Equipment Corporation.

Debugging options to provide a trace of *mbuf* allocation requests and to write protect unreferenced *mbuf* clusters proved useful in tracking down some of the *mbuf* related problems.

Performance Evaluation and Tuning

Work after the initial port focused on evaluating and enhancing performance, decreasing network loading, supporting the User Datagram Protocol (UDP) [3], and supporting the Host Monitoring Protocol (HMP) [4]. Throughput was measured between two VAXes connected to an Ethernet by Interlan controllers. The Ethernet was used to carry traffic for other projects while measurements were made, but this other traffic was light since measurements were taken during off hours. Both VAXes ran with one active user during the tests. Each VAX had a single process running to measure protocol speed, and neither program did any other processing of the information transferred. Initial measurements detailed in Table 1 showed that the BBN code worked at about 60% the rate of the Berkeley code when both kernels set send and receive buffering for the connection to be 2048 bytes.

	bits/second		
	BBN	Berkeley	ratio
750 send to 780	360,088	555,392	0.64
780 send to 750	302,848	531,952	0.56

Table 1: Initial Comparative Performance Measurements

How does one increase throughput? One can attack the problem two ways when one recognizes that the amount of time spent in the network code is the average time per packet times the number of packets serviced. First, we ensure that the operating system efficiently utilizes the CPU when processing a packet. Second, we work to reduce network traffic.

Simple methods to decrease CPU utilization include using registers optimally and choosing good algorithms for manipulating data structures.

After completing some local optimization of this type, a decrease in throughput was seen for small packets. Why? For each packet the kernel sent, it calculated the window that TCP should advertise by walking down a list to count the number of bytes currently buffered. The changes had sped up packet transmission more than packet reception processing, and the receiver's buffer chain got quite long, further slowing down the receiving TCP. Keeping track of the current amount of buffered data and working to keep the list short remedied the situation and brought forth the performance gains that were expected earlier.

After a lot of local optimization, the protocol's implementation was looked at in a more global way using the UNIX utilities *kgmon* and *gprof*. *Gprof* showed what routines consumed most of the time, and from where they were called. *Gprof* allowed the sending and receiving paths to be inspected as a whole. As a result, certain subroutines were turned into macros to take advantage of in-line coding. Wasteful calls to some routines were removed, and some small routines were recoded. The measurements also pointed toward a change that had been tried earlier, but abandoned.

A lot of time is spent producing an acknowledgement (ACK) by the receiving TCP, and processing the acknowledgement by the sending TCP. [5] If fewer acknowledgements are sent,

[3] The User Datagram Protocol uses IP to transport packets, is appropriate for transaction oriented systems, and provides a minimum of protocol overhead. Messages are not guaranteed to arrive at their destination. Data integrity is provided by a checksum.

[4] The Host Monitoring Protocol is meant to be used by network monitoring stations so that operators can find out what is going on in a network.

[5] Let's review the role of the acknowledgement in TCP. A sending TCP moves bytes from the user's address space to *mbufs*. Once in the kernel buffers, the operating system periodically retransmits data until the receiving TCP indicates that it has arrived intact. Upon receipt of the acknowledgement, the sending TCP purges the acknowledged bytes from its buffers.

fewer packets need to be processed by each machine and network load is reduced. However, if ACKs are delayed too much, throughput can be lost due to unnecessary retransmissions by the foreign host. When just every other ACK-only packet is sent, throughput can increase by 20-35% when the data packets are small. When data packets are large, 3-15% of throughput is lost because the sending process is blocked waiting for buffering, and copying new data into the kernel buffers is delayed until space in the send buffers is freed when the acknowledgement arrives.

Subsequent work avoided losing throughput when sending larger chunks of data by estimating the recipient's idea of the usable window and his amount of transmit buffering. The new algorithm assumes that the sending TCP has at least 512 bytes of buffering available and that its Silly Window Syndrome avoidance algorithm [6] does not cut in until more than 50% of the offered window is used. This ensures that the ACK delay algorithm will work well with different TCP implementations. When this work was completed, the effect of delaying ACKs was measured. The results are presented in Table 2.

Seconds to write a buffer 10,000 times

buffer size	maximum number of ACK-only packets delayed					
	0	1	2	3	4	5
1	84	67	63	60	58	57
512	111	97	97	96		
1024	114	115	114			
2048	247	250				

Table 2: Effect of the ACK-delay Algorithm

In our implementation, the default maximum number of acknowledgements to delay has been set to one, to be conservative in communicating with other TCPs. When an ACK is skipped, a timer is started and if an ACK is not sent before the timer goes off, one is then sent. The default time to wait is half a second.

Some time was also spent measuring the effect of Berkeley's trailer protocol and the effect of kernel buffering on TCP throughput. The trailer protocol was turned on and off for both the BBN and the Berkeley implementation. The results of measurements of the trailer protocol are detailed in the Appendix (Table 6, Figure 1, and Figure 2). In general, the use of the trailer protocol provides no performance gains. Measuring the effect of trailers on the application *ftp* demonstrates that the trailer protocol is unnecessary.

Buffering has a large effect on TCP throughput. Both the sender's and receiver's amount of buffering affect the bit rate. When running the benchmarking program, the amount of receive buffering seemed to be more important. Increased kernel buffering for the recipient allows the sender to have several packets in flight. An acknowledgement is sent before the received data is copied into user space. Even if the data will be copied into the reader's buffer immediately, it decreases the advertised window. For this reason, receive buffering larger than the size of the incoming TCP segment allows a non-zero window to be advertised and keeps data flowing smoothly. Increased kernel buffering for the sender allows some data to be ready to be sent upon the arrival of an ACK, and permits full advantage to be taken of the receiver's window. The effect of varying the amount of buffering is shown in Table 3.

[6] Silly Window Syndrome is a condition where transmission is excessively fragmented. Rather than respond to a small usable window by sending a small packet of data, a TCP can perform better by waiting for more data to be acknowledged so that it can send a larger packet. The solution to Silly Window Syndrome reduces message traffic by sending fewer, larger messages.

VAX 750 writing 1k buffers on connection to VAX 780 (bits/second)				
receive	send (bytes)			
	2048	3072	4096	5120
2048		391,961		
3072		602,352		
4096	597,956	712,347	751,559	765,607
5120		712,347		
6144		718,596		

Table 3: Effect of Varying Send and Receive Buffer Allocation

Results of Performance Enhancement

Is it possible to further increase throughput? Perhaps some small gains could be made, but the current protocol machine is well optimized. Current benchmarks produce results detailed in the Appendix (Figure 1, Figure 2, Table 4, and Table 5). These measurements show that the BBN code runs at 92% [7] the rate of the Berkeley code for doing bulk transfer of data over an Ethernet. That the BBN code can handle heavy traffic virtually as well as the Berkeley code is verified by a common application. The BBN TCP/IP can transfer a large file like the dictionary from one VAX to another at 95% the rate of the Berkeley code. The small difference in speed between the two implementations would likely be hidden on a loaded system where the machine must perform other activity as well.

On a long haul net composed of IMPs [8], benchmark results depend on other hosts' traffic more than results obtained on an Ethernet do. Intermediary IMPs may need to spend time to switch messages for other hosts' connections. So, several measurements were taken to try to average the influence other hosts' connections have. BBN TCP/IP can *ftp* [9] the dictionary from one VAX to another at about 102% the rate of Berkeley TCP/IP (see Table 5).

Functional Comparison

Robustness, network and CPU loading, modularity, and supportability are also criteria by which to judge a TCP implementation. We've already discussed modularity and supportability for the BBN code. (One measure of relative modularity is that the Berkeley code uses 124 goto statements to BBN's 9.)

In practice, most network activity is to support remote login and to transfer files. The delaying of acknowledgements previously mentioned reduces CPU usage on the hosts connected by *telnet* connections. [10] It also reduces the amount of work IMPs and gateways have to do. Further traffic reduction on *telnet* connections is achieved by bundling up characters in the operating system's pseudo-teletype code. With BBN TCP/IP, the packet containing the typed character is received in a device interrupt. The packet is processed in a network software interrupt, and the character is passed to the user process. When the character is echoed, it is bundled up with other echoed characters in the pseudo-teletype driver, and one fifth of a second later a group of characters and the acknowledgement are sent toward the user's terminal.

[7] To arrive at 92%, average the figures for writing 256, 512, and 1024 byte buffers on a connection between VAXes on an Ethernet.

[8] An Interface Message Processor is an ARPANET packet switch node. Hosts are connected to IMPs. When a host hands an IMP a packet, IMPs are responsible for transferring the message to its destination. Depending on net topology, several IMPs may be involved along the way.

[9] *Ftp* is a program that transfers files between hosts. Servers have been written for UNIX, VMS, TOPS-20, and other operating systems.

[10] *Telnet* is the standard remote login facility on the Internet. Servers have been developed for UNIX, VMS, TOPS-20, and other operating systems. It is much like the UNIX utility *cu*.

Delaying acknowledgements has a beneficial effect on *ftp* transfers. When acknowledgement delay code was installed, file transfers occurred 5% faster. It has a very good effect on *telnet* connections. Network traffic reductions benefit both hosts and all intermediary gateways and IMPs.

To compare the robustness of the two systems, let's look at the handling of Internet Control Message Protocol (ICMP) messages and routing. The Internet Control Message Protocol defines messages used to signal error conditions. The BBN network code uses ICMP redirect messages to re-route current connections. The Berkeley network code does not re-route current connections. Both kernels add the new gateway to their routing tables, but since the Berkeley code does not re-route current connections, network traffic and machine loading are increased as redirect messages are generated by the gateway for each outgoing packet.

An ICMP source quench message signals the recipient to reduce his message traffic. Berkeley TCP/IP ignores source quench messages. BBN TCP/IP reacts to ICMP source quenches by requiring a larger usable window from the receiver when deciding whether to send a packet. This approach recognizes that the Silly Widow Syndrome and source quenching are related because both involve excessive traffic. As a result, the BBN code uses a single solution for both of these problems.

When a TCP connection is made, a local route is chosen by the kernel. The route describes where to send packets in order for them to arrive at their destination. If a machine has only one interface, this routing decision is limited to a choice of whether to send the message to a gateway and, if so, to which gateway to send the message. If a machine has more than one interface (is multi-homed) the local routing decision becomes more complex. BBN's algorithms for choosing a route are similar to Berkeley's; a path to the destination host is chosen before a path to its network or a path using a smart gateway. [11] BBN's algorithm for choosing a route differs when the source address is known. Then the route must involve going out a network interface with the connection's local IP address so that messages claiming to be from address X really do come from address X.

BBN and Berkeley maintain routes differently. Gateways referred to by routes in use are "pinged". Pinging involves sending periodic ICMP echo requests to the gateway to detect when it goes down. While the gateway is up, the VAX receives echo replies. When a gateway misses too many pings, routes using that gateway are marked down, and connections are re-routed. This maximizes the usefulness of the connection. Rather than constantly retransmitting until a timer goes off and giving the user an error condition of ETIMEDOUT, the information flows another way or, if there is no other path, the user receives an error condition of EHOSTUNREACH. A problem with this strategy is that excessive pinging leads to increased network traffic overhead and increased load on the gateways. Excess ping traffic is avoided by not pinging gateways successfully passing new data for TCP connections, and by only pinging gateways currently used by some connection.

On networks, like the ARPANET, where network level messages state that a machine or gateway has gone down, BBN's kernel treats TCP connections the same way it treats them when gateways "ping out" (are declared down by the pinging algorithms). Berkeley's kernel ignores these messages and assumes that the routing entries for that gateway will be removed manually. In Berkeley TCP/IP, ICMP host dead messages only work when the message refers to the host at the other end of a connection.

In the BBN implementation, when gateways ping out routes using them are not removed from the routing tables. Instead, a timer is set and the route is reinstated after a reasonable period of time. Routing table maintenance thus requires no human intervention and resides wholly within the kernel.

[11] A path to a machine on another network involves using a gateway to bridge the gap between the nets. A smart gateway is a gateway that participates in the full gateway routing algorithms and knows how to get to any connected network, regardless of whether it has a direct connection to the desired network or not.

Other differences can also be noted. The BBN code handles interfaces going down better, notifying connections using that address as either the local or foreign address. The Berkeley code notifies just connections with that foreign address.

The BBN raw interface is more comprehensive. Using the BBN code one can monitor all IP packets without resorting to monitoring at the device level. If one is interested in just TCP, UDP, ICMP, or HMP packets, then by opening a raw socket for that protocol one takes advantage of address sorting done by the kernel and one receives only packets destined for specific addresses or protocols. In comparison, the Berkeley kernel passes just a small subset of ICMP packets to the raw interface for the Internet domain.

The BBN kernel handles multiple bytes of urgent data properly. Berkeley saves just a single byte of out of band data for the user. Multiple bytes of urgent data are used for the break and sync options of the TELNET Protocol.

The BBN kernel provides *ioctl* calls so that a program may set the amount of send and receive buffering it would like the kernel to provide. No such *ioctl* is implemented in the Berkeley kernel. Additional *ioctl* codes provide for setting connection initialization timers, inactivity timers, and the TCP push option. In general, the BBN implementation gives the programmer greater control over how connections are defined.

Summary

BBN TCP/IP provides throughput comparable to Berkeley TCP/IP. Considering the desirable goals of reduced host and network loading, easy program maintenance, and correct handling of as many conditions as possible, we feel that BBN TCP/IP provides a superior protocol layer for 4.2BSD.

Appendix A

TCP/IP Benchmarks for 4.2BSD

All measurements were taken April 12, 1984 so that all kernels would be presented with the same network environment. Send and receive socket buffering was set to 4096 bytes for all kernels. The BBN TCP/IP delayed at most one consecutive ACK-only packet.

Tests done on the IMP network used BBN-NET, the test-bed for the ARPANET. Each VAX was connected to an IMP. There were two shortest paths between these IMPs, and each involved one intermediary IMP. Therefore, each packet passed through 3 IMPs to reach its destination.

In Table 6, the column gain is the ratio of bit rate with trailers to the bit rate without trailers. A gain of one implies that using trailers neither sped up nor slowed down the transfer of information.

		BBN	Berkeley	
direction	file size	seconds	seconds	ratio
750 to 780	201,039	11.12	10.87	0.97
780 to 750	201,039	10.61	9.95	0.93

Table 4: Ftp between a VAX 750 and VAX 780 on an ethernet

			BBN	Berkeley	
	direction	file size	seconds	seconds	ratio
April 12	750 to 780	201,039	50.92	48.72	0.95
	780 to 750	201,039	50.62	54.34	1.07
May 6	750 to 780	201,039	52.93	53.39	1.00
	780 to 750	201,039	52.60	54.91	1.04

Table 5: Ftp between a VAX 750 and VAX 780 over IMPs

direction	file size	seconds	gain
Berkeley 750 to 780	201,039	10.88	0.99
Berkeley 780 to 750	201,039	9.71	1.02
BBN 750 to 780	201,039	11.39	0.97
BBN 780 to 750	201,039	10.43	1.01

Table 6: Ftp between a VAX 750 and VAX 780 on an ethernet using trailers

VAX 780 sending to VAX 750 over an Ethernet

April 12, 1984

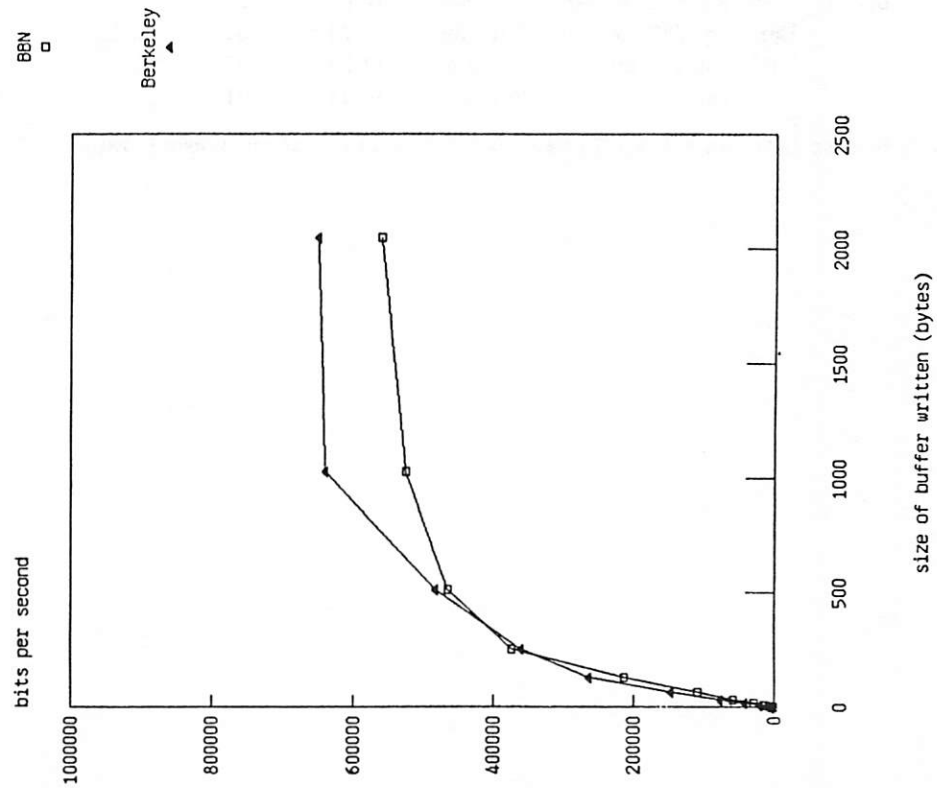


Figure 1

VAX 750 sending to VAX 780 over an Ethernet

April 12, 1984

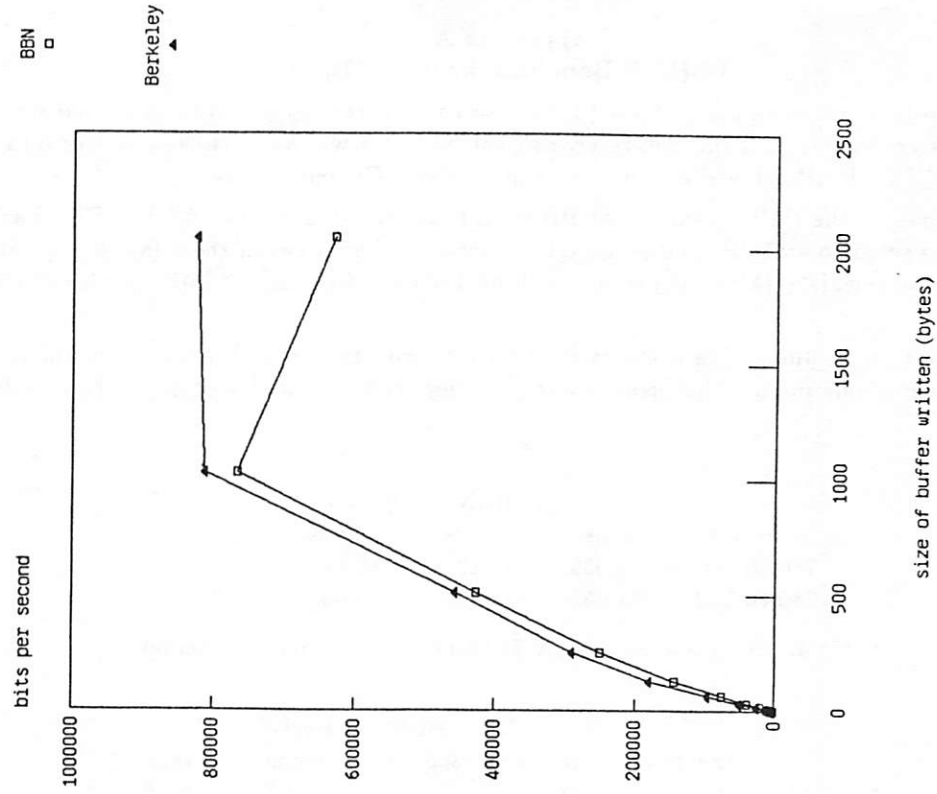


Figure 2

VAX 780 sending to VAX 750 over Ethernet using Berkeley TCP/IP

April 12, 1984

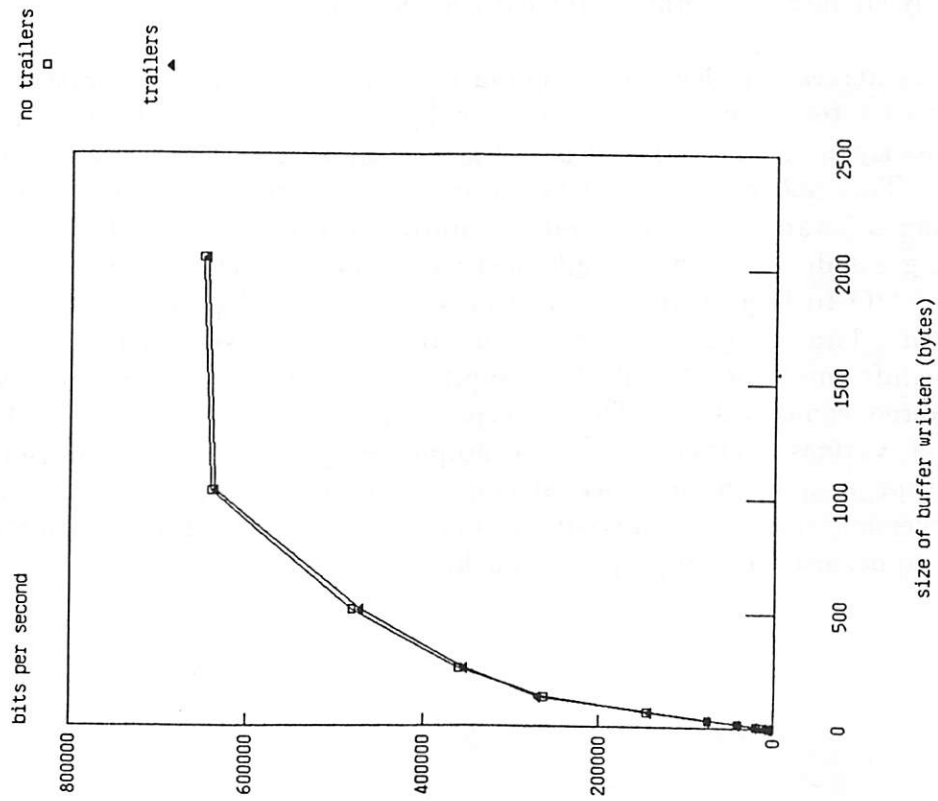


Figure 3

VAX 750 sending to VAX 780 over Ethernet using Berkeley TCP/IP

April 12, 1984

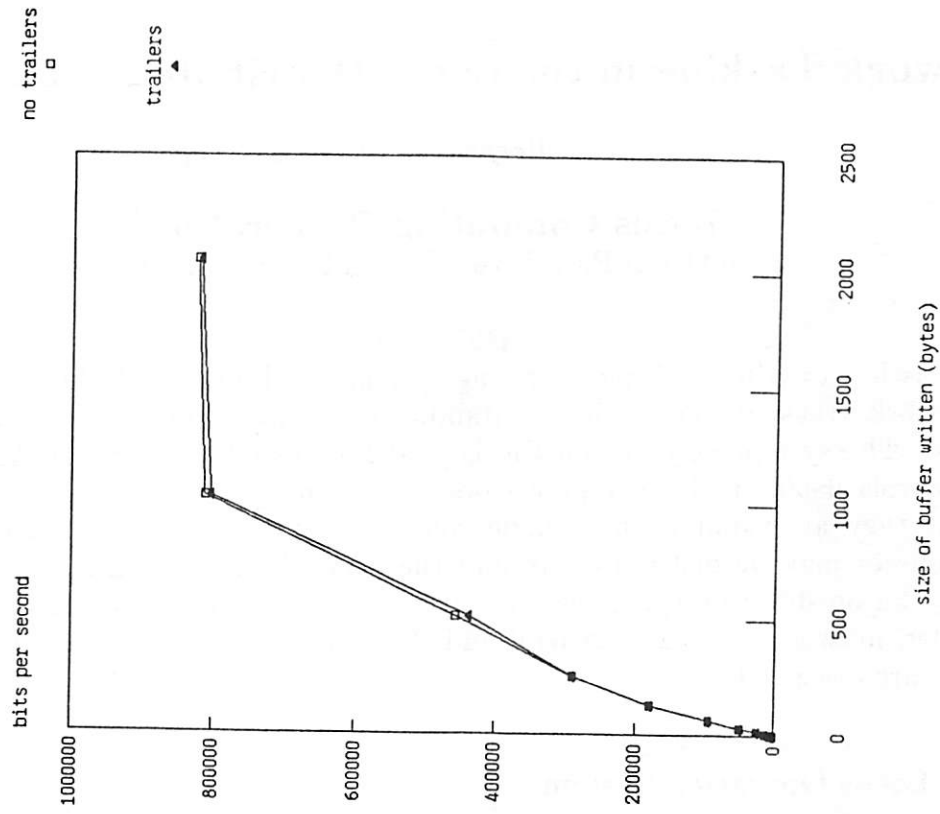


Figure 4

Network Tasking in the Locus Distributed Unix System

David A. Butterfield and Gerald J. Popek

Locus Computing Corporation
3330 Ocean Park Blvd., Santa Monica, CA 90405

ABSTRACT

Locus is a distributed Unix operating system which provides a high degree of network transparency while maintaining excellent performance characteristics. The system executes on the Digital Equipment Corporation VAX, the Motorola 68000, and other processors. Users and programs have the illusion that they are running on a single computer system, even though their file resources may be distributed around the network and cooperating processes may be on different machines, or may even move during execution. In this paper, network tasking in general and the tasking aspects of Locus in particular are discussed.

1 The Locus Operating System

The Locus operating system is a distributed version of Unix, with extensive additional features to aid in distributed operation and support for high reliability and availability behavior. Many of these facilities are of substantial value even when Locus is used on single, standalone computers. Locus provides a surprising amount of functionality in a reasonably small package. The system, while somewhat larger than standard Unix, is significantly smaller than other extended Unix systems.

The system makes a collection of computers, whether they are workstations or mainframes, as easy to use as a single computer, by providing so significant a set of supports for the underlying network that it is virtually entirely invisible to users and applications programs. This *network transparency* dramatically reduces the cost of developing and maintaining software, and considerably improves the user model of the system. It also permits a great deal of system configuration flexibility, including diskless workstations, full duplex I/O to large mainframes, transparently shared peripherals, and incremental growth over a large range of configurations (from one workstation to a large network including mainframes) with *no* effect on applications software required to take advantage of the altered configuration. This transparency is supported even in a heterogeneous network of various cpu types. For example, if a user or program running on one machine type in a Locus network attempts to execute a program for which the only available version runs on a different machine type, Locus will automatically cause that execution to occur on the appropriate machine, completely transparently to the caller.

Locus has been in operational use for over two years on DEC VAX-11/750 computers. Systems containing as many as 17 nodes and three cpu types have been operated. Nodes in a Locus network may communicate over a variety of media; standard ten megabit Ethernet is common.

This paper first briefly reviews the major characteristics of Locus, and then concentrates on the remote tasking aspects of distributed operation. Remote process handling is described both philosophically and concretely as implemented in Locus. The Locus user model is presented, and some of the difficulties encountered in implementing it are mentioned. We conclude with comments on the feasibility of implementing the network process model and experiences with actual applications using the Locus implementation.

1.1 Transparent Distributed Operation

The power of network transparency and compatibility with an existing operating system environment can hardly be overestimated. It is commonplace to observe that distributed computing is very important, both in the near and longer term future. It is less well understood that unless a wide collection of distributed system application software becomes rapidly available, the growth of distributed computing will be severely retarded.

The Locus philosophy of very extensive transparency, together with Unix compatibility, even in distributed mode, means that all programs which run on a single machine Unix system will operate, without change, with their resources distributed. That is, one can take a standard application program, set a few configuration parameters, and cause it to execute with parts of the program running on one machine, parts on other machines, and both the data and devices which it accesses distributed throughout the Locus network, all without change to the application. This ability means that there is immediately available a very large collection of distributed application software, the necessary prerequisite to commercially effective distributed computing. Locus also includes facilities to control where data is located and which users and machines can access data, peripherals, cpus, and logical resources.

1.2 High Reliability

Reliability and availability are key to applications envisioned for Locus for two general reasons. First, many of the applications themselves demand a high level of reliability and availability. For example, the effect of a machine failure for an office system user is the equivalent of randomly locking that user out of his office at a random point for an unknown duration, an unacceptable situation. Database requirements for reliable operation are well known. Second, the distributed environment presents serious new sources of failures, and the mechanisms needed to cope with them are more difficult to construct than in single machine environments. To require each application to solve these problems is a considerable burden. Locus provides advanced facilities for both the local and distributed environment which operate transparently. These facilities include a file system which guarantees that, even in the face of all manner of failures, each file will be re-

tained in a consistent state, either completely updated or unchanged, even if alterations were in progress at the time of failure. This *commit* mechanism involves no additional I/O and so performs very well.

One of the important reliability and availability features of Locus is its support for automatic replication of stored data, with the degree of replication dynamically under user control. Loss of a copy of a replicated file does not affect continued operation, and Locus assures that the old version will be automatically brought up to date when it becomes available. A general transaction mechanism is provided that includes such advanced features as support for nested transactions, even when the transaction itself is distributed. Measurements demonstrate that this facility performs very well. Redundancy checks are included to avoid propagation of errors.

1.3 Unix Compatibility

Locus has been derived from Unix by successive modification of its internals and considerable extension. For virtually all applications code, the Locus system can provide complete compatibility, at the object code level, with both Berkeley Unix and System V, and significant facilities are present to provide high degrees of compatibility with other versions of Unix as well. As a result, one routinely takes unrecompiled load modules from other Unix systems and runs them unchanged on Locus. The process of converting an existing Unix system to Locus is designed to be a largely automated step of removing the Unix kernel and associated software, inserting the Locus kernel with its supporting system software, and reformatting secondary storage to add the reliability facilities and functional enhancements.

1.4 Heterogeneous Machines and Systems

Locus contains a set of facilities that permits different cpu types to be connected in a single Locus system. A high degree of transparency is supported across the various machines. For example, attempts to execute programs prepared for different cpus will automatically cause execution on the appropriate target machine, all transparently to the calling program or user. Such sophisticated transparency is accomplished in a manner that does not add complexity to the user view of the system.

2 Introduction to Network Tasking

There were two very strong considerations in choosing a user model of network tasking for Locus. One of these considerations was that Locus is a Unix system, and as such it should be compatible with other Unix systems. This means that portable C programs from other Unix systems should run unmodified in the Locus network environment. It was considered desirable for these programs to be able to take some advantage of the network, even if they were naive about its existence. It was also considered desirable that it be easy to add minimal code to an existing program which would allow it to make extensive use of the network.

The other major consideration was the overall Locus philosophy of *network transparency*, which requires that processes operate and interact with files and other processes across the network in the same way and with the same effect as locally. Adhering to this philosophy allows the user model to be considerably simpler than it might otherwise be, makes development of distributed algorithms easier, and allows graceful reconfiguration or degradation of distributed groups of cooperating processes when some nodes of the network are unavailable. (References to more complete discussions of various aspects of network transparency may be found in the Bibliography.)

Thus, the major considerations when choosing a network model were that it fit well into the Unix model, providing compatibility, and that it allow local and remote processes to be manipulated in the same ways, providing network transparency. Other important desirable characteristics were ease of user control, high performance, and retention of local autonomy of individual nodes in the network.

2.1 Network Tasking in Locus

The major Unix process control functions *fork* and *exec* have been extended in an upward compatible way. Recall that *fork* creates a new process, running the same program image as the caller, and that *exec* replaces the code and data of a running process with a new program and data image. In Locus, both of these calls have been extended for the network. For compatibility, the system calls look the same as in Unix, but under certain circumstances they operate over the network. *Fork* can cause a new process to be created locally or remotely. *Exec* has been extended to allow a process to migrate to another site as it replaces its image.

The decision about where the new process or new image will execute is made by examining a list of preferred execution sites specified by information associated with the calling process. This information is inherited by child processes when a process forks, and can be set dynamically by a *setxsites* system call. By issuing *setxsites*, a running program can specify where its subprocesses and future images will execute. An interactive user can set the site of execution of subsequent programs by giving a command to his shell, which will issue the *setxsites* call.

Use of an additional system call to set execution site information was chosen rather than adding arguments to the process calls for two reasons. First, by not changing the existing system call interfaces, programs written without knowledge of the network continue to work, and in fact may have their execution sites selected by their parents. This aspect is especially valuable when source code is not available. Second, it was considered desirable to separate those functions which affect the semantics of a program from those functions which are performance optimizations. With few exceptions, the site where a process runs does not have any effect on the results that the process computes.

A new system call, *migrate*, has been added to permit a process to change its site of execution while in the midst of execution. A long-running process might choose to do this, for example, if it received a signal telling it that the current site was about to be taken down.

Another application for process migration is load leveling. Process migration can be induced externally by sending a new signal whose default action is to migrate the process. If the user code of the process does not catch or ignore the signal, then the process will move to a new site. Using this mechanism, a load leveling daemon can monitor loads on various machines on the network and cause long-running cpu-intensive processes to move to less loaded sites.

2.2 Local Autonomy of Sites

In some Locus networks, it may be desirable to restrict the use of various machines in the network for use by only certain users. For instance, some machine may be reserved for only a particular group. The Locus network tasking protocols are designed so that each site in the network makes the final decision about whether to accept a process which is attempting to migrate to that site, and whether to create a process when a fork to that site has been requested. The site can implement whatever permission checking it deems necessary. In this way, the local autonomy of the site is preserved with respect to cpu resources expended upon user processes.

In addition to the Unix limitation on the number of processes a user may have on a site, Locus associates a site permission mask with each process (inherited by children) which specifies to which sites the process may perform network process creation or migration operations. This permission mask is set at signon time by *login*, on an individual user basis.

2.3 Integrated Fork/Exec Call

In typical Unix programs which create subprocesses, very often the *fork* call is quickly followed by an *exec* call. In Locus another new system call, *run*, has been added. *Run* is intended to achieve the effect of a combination of *fork* and *exec*. *Run* creates a new process and invokes a new program image in it before returning to the user code of the child process. In this way, the costly overhead of the *fork* system call can be avoided, without sacrificing machine independence.

To be useful in most situations, *run* takes as an argument a structure specifying certain operations to be performed by the kernel in the child process before the new program image begins execution. The permitted operations are those found between the *fork* and the *exec* in typical Unix programs. These include operations which close or duplicate file descriptors and change signal actions.

Run can create the child process locally or remotely, just as *fork*, and since it invokes a new image, may operate across different cpu types.

2.4 Local File Names

There are a few cases where it is useful, by convention, to associate certain files with a given site in the network. In Locus, it is common for there to be a directory for each machine in the global tree structure, e.g. */mach_i*, */mach_j*, and so on. One may even wish to make these directories local to their associated machine by making them mounted file systems. Such directories can, if the system administrator chooses, be used in Locus to hold the following types of items:

- a. Per-site utility files: these include boot command sequences, local accounting data, and so on;
- b. Temporary scratch storage: the files typically put in */tmp*;

Thus, for example, the globally transparent name for the temporary disk storage file *save* on site *payroll* would be */payroll/tmp/save*.

However, in Unix, by convention, the names used for the few purposes mentioned above do not include a *machine_name* directory as part of the path name. Therefore, Locus includes an *alias* mechanism in the file system to map these old names to the new places in the name tree. Each process has associated with it a context variable which contains *machine_name*. Under normal conditions, if a process issues one of the old names, say */etc/utmp*, then the name which would actually be referenced might be */machine_name/etc/utmp*.

There are very few file names handled in this way; typically a dozen or so, primarily only system utility related items, of little or no consequence to most users.

The context variable is initialized to be the name of the machine where the user logs in. Its value is inherited when a parent creates a child process, and is unaltered by migrating a process. Thus, members of a process family, even if distributed among a number of machines, experience the same environment, and process migration need not alter the effect of a program's execution.

Through this alias mechanism, Locus accommodates the few cases in Unix where conventions embodied in a small number of old, existing programs, together with performance considerations, mandate that a few names not be interpreted in a global, transparent way. Since all resources can still be naturally accessed by their basic, globally transparent names, it is hoped that most of these few vestiges of single computer operation will eventually atrophy.

2.5 Heterogeneous Locus Networks

In Locus networks with more than one cpu type, it is clear that *fork* and *migrate* can be done only between matching cpu types, since the process retains substantial internal state information, such as registers, stack frames and so forth, not to mention the instruction stream. The source and destination site cpu types of an *exec*, by contrast, may differ, because most of the machine dependent information is reset.

When a load module is selected for execution, it is examined to determine on what cpu type it will run. The execution site list for the process is searched until a site of the appropriate type is found, or an indicator is reached that the system should simply choose an appropriate site.

Networks with heterogeneous cpus present a special problem for storing of load modules. Since the naming hierarchy is globally known throughout the network, there can exist only one object called, for example, */bin/cat*. In a homogeneous network of VAXen, this file would contain a VAX load module for the Unix *cat* program. In a homogeneous network of 68000s, it would contain a 68000 load module for the Unix *cat* program. In a heterogeneous network containing both VAX and 68000 processors, the obvious problem arises.

One might say that there is no problem here at all, that whichever program is placed into the file, *cat* will still work from any site in the network. Suppose that */bin/cat* contains a VAX load module. In this case, if the file is executed from a 68000, the system will notice that the file cannot run locally, and will find a VAX in the network to run the program.

Unfortunately, this would require that there always be a VAX up and available in the network, in order for *cat* to run. Further, this requires that every execution of *cat* from a 68000 be performed remotely, rather than locally.

The Locus solution is a *hidden directory*. A *hidden directory* is a special directory used for load modules; one load module for each machine type in the network is placed into the hidden directory, and when an attempt is made to execute the directory, the system automatically selects one of the load module files in the directory to run. In the example above, */bin/cat* would be a hidden directory containing two files: */bin/cat/vax*, and */bin/cat/68000*. Execing */bin/cat* would select one of the load modules from within the directory and execute it.

The directories are termed *hidden* because opening the directory to read it (e.g. by executing *size /bin/cat*) also selects the appropriate component from the directory and opens that file. Thus, the *hidden directories* appear to be simple files to most application programs. This is compatible with Unix systems.

A process has associated with it a list of names which are used to choose the components from hidden directories. This list is inherited by children from their parents, and may be set explicitly. It is initially set to select components for the type of site on which the user logged in.

Note again that hidden directories are needed only in heterogeneous Locus networks, to collect together the load modules for the various machine types in the network. In homogeneous networks, there is only one load module, so it may simply be installed as a file.

2.6 Network Tasking Applications

One application of remote tasking is load leveling. As was noted in a previous section, load leveling can be performed automatically by a daemon process which looks for cpu bound processes and unloaded sites, and asks the processes to migrate there.

A simpler form of load leveling may be performed by the shell. If the user has requested this option, the shell can keep track of the load on various sites and automatically execute the user's commands on the least loaded site. The mechanisms required to implement load leveling were almost trivial to construct under Locus.

Another application of remote tasking is the distribution of programs which have inherent parallelism. The *make* program is an example. *Make* has been modified so that multiple machines may simultaneously participate in making a final target which requires several intermediate targets. With two machines participating, the compile time of a major program is cut nearly in half. The changes required to *make* were quite simple.

2.7 Implementation Challenges

Implementing transparent remote tasking presented several difficulties. A major difficulty was *process tracking*. It is sometimes necessary to find a process in the network to deliver a signal or to inform it that its child or parent has died. To do this without having to search the entire network requires that the system keep track of processes when they migrate about. The process tracking code must be extremely careful when processes are moving around while delivery of a signal is being attempted, so that the signal is delivered reliably.

Another major effort was the code which implements file sharing among related processes on different sites. Unix file sharing semantics require inherited file descriptors to share read/write pointers among the processes involved. Locus implements these semantics across the network; the most common cases execute as fast with the processes distributed as when they are collected on a single site.

2.8 Network Tasking Experience

Our experience with transparent tasking has been an unqualified success, from the user's point of view. As illustrated above, the job of building software to execute in a distributed environment has been substantially eased, and the simplification of the user interface has been found to be quite valuable. As a further example, some time ago, a demonstration of Locus on a 68000 workstation connected to a larger VAX Locus network was being given. The demonstrator edited a program source file for a given language, and then compiled and ran it, piping the results to another program running on his workstation. Only afterward did he realize that there was no compiler for the language on the workstation; the compilation and subsequent program execution took place, transparently, elsewhere, on a VAX.

However, from the Locus implementors' point of view, the design and development of transparent tasking required considerably more effort than had been originally contemplated. Of course, the existence of a very high degree of transparency in the basic Locus system was an essential prerequisite. Without it, transparent tasking is obviously not feasible. Nevertheless, transparency still had to be extended to process relevant issues, such as shared file descriptors, signals, and so on. Also, given how much had already been accomplished, the opportunity to further extend the Locus transparency mechanism to heterogeneous cpus was especially attractive. Although heterogeneity presents significant additional problems, and complete solutions are not feasible, nevertheless a great deal can be accomplished, and the result so obviously frees the user from so much unnecessary complexity in his environment that the cost of development has clearly been justified.

3 Conclusions

The Locus network tasking implementation permits one to execute programs at any site in the network, subject to permission control, in a manner just as easy as executing the programs locally. One can dynamically, even just before process invocation, select the execution site. No rebinding or any other action is required. The mechanism is entirely transparent, so that existing software can be executed either locally or remotely, with no change to that software. This facility makes execution of a program remotely, as well as the construction and execution of distributed programs, quite straightforward.

Locus tasking greatly simplifies the development of software for a distributed environment. Load leveling is simple to accomplish. Automatic selection of the site of execution as a function of needed resources, including cpu characteristics, is also done automatically. Execution of programs with inherent parallelism, such as *make*, can be done on multiple machines simultaneously with little change to the original, single machine program.

Now that a highly transparent distributed computing environment is available for everyday use, those of us who have become accustomed to it take it for granted. Those who still remember working in a conventional networking environment wonder how they ever got along without transparency, and generally can't conceive of going back, any more than they would willingly give up interactive computing, screen editors, or virtual memory.

4 Bibliography

The Locus Distributed System Architecture, Locus Computing Corporation, Santa Monica, California, 1983.

Popek, Gerald J. and Walker, Bruce J. *Network Transparency and its Limits in a Distributed Operating System*, UCLA Computer Science Department Technical Report CSD840228, Los Angeles, California, 1984.

Walker, Bruce J. *Issues of Network Transparency and file replication in the Distributed Filesystem Component of Locus*, UCLA Computer Science Department Technical Report CSD830905, Los Angeles, California, 1983.

Walker, Bruce J. *The Locus Distributed Computing Environment*, 1984 IEEE Aerospace Applications Conference Digest, New York, New York, 1984.

Project Athena

James Gettys

Digital Equipment Corporation
MIT Project Athena
decvax!mit-athena!jg
jg@mit-athena.arpa

Project Athena is a joint project of MIT, Digital Equipment Corporation, and IBM to harness the power of the computer for education. Over the next five years, approximately 1600 VAXTM based personal computers and work stations will be installed at MIT for an experiment in undergraduate education. It is hoped that the computer will become a standard tool in the MIT Curriculum. Besides the donation of equipment, this project involves the placement of full time engineers of both corporations at MIT.

Athena is based on the principle of "Coherence"; faculty and students should see the same environment. Toward this end, all equipment will run the same software, currently based on 4.2BSD UnixTM. Sharing of programs and data are an essential goal of this project.

The project is organized into two phases. In the first phase, a fraction of the equipment is being installed permitting curriculum software development to begin and operational problems to surface in time to be remedied before scaling of the project to its final size in the second phase.

A campus network is an important part of this system. From the outset, software has been installed over the network. Building a complete network also involves installation of cable all over campus; ultimately it involves wiring every office, dormitory room and class room and therefore becomes as pervasive as the telephone. While detailed plans are still being made, a sketch of the network architecture can be given and the justification. A fiber spine network is being installed around campus using redundant Proteon Ring technology. Local clusters of machines are being connected to the spine using small gateway machines to provide isolation of local machines and the campus wide network.

1. Organization

At the present time, we have (in one location), employees of MIT, Digital, and IBM. It is planned that approximately five full time engineers of Digital will work on Athena, and approximately twenty MIT employees for the duration of the project. IBM will provide a similar number of engineers as Digital. MIT is also providing physical space, power, and cooling for the equipment and support for faculty and student curriculum development projects. Digital hardware will be used primarily in the School of Engineering, while IBM hardware will be used in other schools. A total of approximately 2500 machines will be installed.

2. Coherence

There are three major problems involved in building large distributed systems. These are:

- 1) Scale.
- 2) Differing machine architectures.
- 3) Diversity of operating systems.

VAXTM is a trademark of Digital Equipment Corporation.

UnixTM is a trademark of AT&T Bell Laboratories.

Athena will push the first two of these problems. Unix permits us to handle the machine architecture problem, while presenting a common environment to the user due to its transportability. MIT feels it vital that it not be tied to a single vendor's operating system or machine architecture.

It is not unusual for a student of nearly any department to see as many as five different operating systems in one semester glimpses during his undergraduate career. If the computer is to be an integral tool for all students, the student must have opportunity to learn its capability rather than re-learn basics each semester. It is also highly desirable that people be able to take advantage of other people's work. Data or programs should be able to be easily shared, limited only by their "usefulness" rather than their availability.

The first obvious step taken in this area is to use one operating system, independent of the underlying hardware. Berkeley Unix is the obvious choice at this point in time due to its networking, IPC and performance advantages. In Athena, we hope to remove the "incoherence" of multiple operating systems, so that the transfer of information and programs will be limited only by their usefulness and by individual privacy considerations.

One of our goals is to augment the Unix tool kit philosophy with more general mechanisms for program and data sharing. This is made yet more important due to architectural differences between machines. In order to fully share data generated on differing machine architectures, interchange of the data must be made easier. It is not always reasonable to convert data to an ascii representation for interchange.

3. Machine Model

The "station" or personal computer will eventually be completely operated by the user. The basic presumption of a minimum machine includes the following points; nothing here is out of the ordinary:

- Approximately 1 MIP compute.
- Large address space machine.
- Virtual memory system.
- 1 MB or more physical memory.
- 1000x1000 display.
- High speed network interface.
- Mouse as principle pointing device.
- Certainly local floppy disk, possibly hard disk.

Other optional equipment that may exist on some machines include:

- Color display.
- Tablet.
- Data acquisition hardware.
- Video disk players for CAI.
- Disk drives.

Central services, administered by MIT, eventually available on the network will include:

- Tape drives
- Printers.
- Mass storage devices.
- Campus Network.
- Compute servers (large mainframes).
- Data base and name servers.
- Mail servers.

4. User Environment

Athena is an educational experiment. During the course of a day, a student will attend classes, work in libraries, and study in their living groups. Faculty and staff, however, are expected to do most of their work in their offices, on the same machine, day in and day out. To serve the educational goals of Athena, a good fraction of machines must therefore be available in public or semi-public areas of classrooms and laboratories.

One attraction of large time sharing systems has been sharing. For example, people share common file systems and mail services and most people are able to ignore operational problems such as backup. Non computer expert people have been able to ignore difficult system maintenance problems. Extraordinary uses of such computers has always been difficult, however, as too many other people are affected by such use. Time sharing computers always seem slowest just when you need them most, as any MIT student will tell you at the end of the semester.

The attraction of personal computers is that you "control" them and that once paid for have little incremental cost of ownership. They are also typically located in a personal workspace and may have special devices attached or confidential data stored on removable media. To date, they have been much slower than the large time sharing systems, but they are completely predictable in response. These are strong arguments for autonomy.

The biggest problem with current personal computers is that there is little opportunity to share programs or data beyond that of interchanging floppy disks. Another major problem with completely autonomous personal computers in a public setting is the probability that a previous user has made the software on the system unusable for some reason. Software distribution is a constant headache.

We distinguish between public machines and "personal computers". The public machines, or "stations", cannot be assumed to retain state between use, and must be easily restored to a known state. A student or faculty member should be able to use a station with little concern that a previous user has laid a trap for the unwary, either intentionally or unintentionally.

Some private machines, "personal computers", however, need to be capable of functioning independently of the network to permit faculty or non-residential student use at home. Completely confidential material is best dealt with on a autonomous machine.

General services will be administered by MIT and in the long term may evolve to reach a larger audience than Athena. These logically include file, print, mail, backup and other I/O services. There must also be provision for non-MIT provided services as computer networks continue to evolve. Note that the model of "personal computers" does not require subscription to these services, though Athena machines will by their nature use them heavily. Others should not be required to pay for services they do not want or use. Athena will be a success if these differing advantages of autonomy and sharing can be combined in a single user environment.

5. Status

The project began about a year ago. In the first phase of the project over sixty Vax 11/750's with VAXstationTM 100 graphics displays will be used as temporary "models" of single user machines. Over half of these machines are

VAXstationTM is a trademark of Digital Equipment Corporation.

now on campus. The 11/750's are expected to support six people each. This permits faculty and students to begin development of software in an environment that resembles the target environment. It is expected that these machines will form the basis of the server configurations in the future as single user machines replace them. We expect initial Microvax I systems in June.

Seventy six proposals for use of Athena resources were submitted by faculty and students in December. Forty six projects were funded. Many of these projects explore novel approaches to education. The first set of machines were turned over to faculty and students for use in early March. Over five hundred people are now registered in our user data base.

6. Network Architecture

While the network is essential to Athena's success, it is part of a more general problem facing computing at MIT. As the network represents a large investment in physical plant and a general facility the campus will depend on, serious reliability, security, and performance considerations become important.

The goal of the MIT network is to transport packets of multiple protocols anywhere on the MIT campus. This network might be termed a "backbone" or "spine" network. Protocol families currently in use at MIT include TCP/IP (being used by the Laboratory for Computer Science (LCS) and Athena), CHAOS protocols (currently the most widespread at MIT, with over 150 existing nodes), and DECNET, with other protocols expected over the life of the project. The campus network is responsible only for transport of packets from one machine to another machine using the same protocols. With already large numbers of network hardware technologies available, the cost of wiring the entire campus for each of them becomes prohibitive.

No user machines will connect directly to the "spine". Local networks will be connected to the "spine" network via gateways. Prototype versions of this gateway technology has been in routine use in MIT LCS for some time for TCP/IP and CHAOS protocol families. These machines are currently based on PDP-11/23's or 68000 processors though higher performance versions are being explored. This work was done in the Distributed Computer Systems group of MIT's Laboratory for Computer Science.

The first spine link will be installed this summer between LCS and E40. This run is somewhat over one mile in length and will connect the initial clusters of machines. It will use proNET™ dual redundant ring technology over fiber optic cable. With time, this spine network will be extended over the length of the MIT campus.

There is a difficult problem reaching many of MIT's fraternities, some of which are across the Charles River in Boston. There are a number of possible solutions to this problem, ranging from leased telephone lines, laser links, microwave links, to arranging for a conduit across the Harvard Bridge, which will be rebuilt over the next several years. No firm plan is in place yet for the solution of this problem; trade offs of cost versus bandwidth are being studied.

7. Unix Changes

There are a set of problems which must be solved for this environment. Most of these are obvious, some of them are less obvious, and are caused by scale and distribution problems.

ProNET™ is a trademark of Proteon, Inc.

Work is underway to provide an authentication server / user database server system. As an interim measure, we are requiring user id and login names be unique across all Athena machines. Our current user community includes several hundred people and can be expected to grow quickly. Various Unix utilities begin to break with scale ("ls" for example) and will have to be reworked. The current 16 bit UID in Unix may have to eventually be enlarged. The data base itself is already in use; yet to be done are the server / client programs for remote access to the data base. A student or faculty member will be able to register himself without administrative intervention very soon.

Major amounts of work will have to be done on mail. Delivery to a single user machine may be delayed indefinitely as there is no guarantee that the machine is powered on or working. Students and faculty should be able to access their mail easily from anywhere in the network and are no longer tied to single machines. We are exploring approaches similar to Grapevine†.

Even in this environment, particularly initially, resources are not unlimited. The principle problems will have to do with disk and paper usage. The 4.2BSD quota mechanism will probably have to be somewhat extended for use in file servers. A contemplated change is to have group as well as user disk quotas on file servers. Joint projects should be able to draw on joint disk resources. The fact that people can belong to multiple groups in 4.2BSD simplifies the data sharing problem greatly. The problem of limiting or charging for paper has not been looked at yet.

Backup is of serious concern. The phase one DEC hardware has over 40 Gigabytes of storage on it. Our temporary solution has been to dedicate most of the disk on one sixth of the machines to automated backup over the network. Magnetic tape, even at 6250 BPI, requires too much human intervention to be satisfactory. The cost of a tape library becomes very large when scaled up to this level, and the administrative burden is very large, particularly due to the security problems of conventional backup. To restore files now requires manual intervention of a system administrator to ensure that only data belonging to a user is restored. We cannot afford to provide such service to a huge user community, nor do we want the bureaucracy that goes with it. Backup should ideally enforce the normal system security mechanisms. It may be that replication of files in multiple locations is the only satisfactory long term solution. Writable video disk might provide a more conventional solution, if available in time. At the moment, we only claim to be able to restore from hardware or system software failure; no mechanism is in place to provide restore services for user mistakes.

Graphics, window systems, and the human interface to them is my personal area of interest. Despite Unix's transportability, it is unrealistic to believe that all software can be run on all machines. For example, a mainframe or array processor is an expensive resource that cannot be provided in each person's machine. It may also be that a person wishes to provide a service to others while not providing the software to them. None the less, such servers should be able to transparently manipulate the user's display. To complicate the issue, we must face multiple different displays of different architecture and capabilities early in this project. Work in this area is just beginning.

Maintaining 2500 completely independent machines will not work either. In particular, the public machines must be able to be put into a known state easily or there is no guarantee that any piece of software will work at any given time.

† Birrell, A.D., Levin, R., Needham, R.M., and Schroeder, M.D. Grapevine: an exercise in distributed computing. *Comm. ACM* 25, 4 (April 1982), 260-274.

Conventional updates become completely impossible. Even if each machine only takes one hour of a skilled person to update this translates to a man year. Some sort of distributed file system is essential on this ground alone.

8. Summary

There are only a few networks in existence of the same size as the one contemplated at MIT. If Athena is a success, it is expected that the size of the network could grow another factor of two or three soon after the end of the project. Only the Xerox, DEC, the Arpa Internet networks are of the same scale. Of these, only Xerox's network has the kind of functionality that we are looking for. Both Digital's engineering network and the Arpa Internet are loosely coupled networks of completely autonomous machines. We are looking to combine the community of the time sharing environment without losing the attractiveness of personal computers.

TEMPO

A Network Time Controller for a Distributed Berkeley UNIX (*) System†

Riccardo Gusella and Stefano Zatti

Computer Systems Research Group - Computer Science Division
EECS Department - University of California, Berkeley

Abstract

TEMPO keeps the clocks of computers in a local network synchronized with an accuracy comparable to the resolution of each individual clock. In a loosely-coupled network the machines can only compute the differences between the times of their clocks. A new algorithm has been devised to perform this measurement; a protocol based on it can adjust the clocks by means of a new system call that has been added to the kernel of the Berkeley UNIX 4.2BSD operating system.

Several experiments show that a *total quasi ordering* can be based on the unique network timing maintained by the service.

Introduction

The abstract concept of *time*, and the problem of measuring it, have always been one of the major concerns of human beings. Time has been related to the declination of the sun, the phases of the moon, the position of the stars. The search for a simple device able to measure this physical quantity has been the next logical step. From the sundial to the hourglass, from the mechanical to the atomic clock, there has been a constant improvement in the accuracy of this measurement. But even in the days of high technology, the problem of keeping geographically dispersed clocks well synchronized has not found a satisfactory solution. If the events of interest occur at the rate of the passage of busses at a bus stop, then the precisions of the driver's and the passengers' watches may be sufficient to coordinate the actions of all the parties involved. When,

on the contrary, events follow each other at a much higher rate, then the precision and the stability of standard quartz clocks may not be enough to assure coordination of activities in a complex system.

All computers utilize one or more quartz clocks to synchronize the internal activities of their constituent parts. One of the clocks, which consists of an oscillator and a counting device, is often used to provide the *time-of-the-day* service. The oscillator generates a periodic waveform at a uniform rate, while the counter records the number of elapsed cycles. When the counter reaches a predetermined value, an interrupt is generated which, in general, causes the software to increment an internal variable accessed by local programs. Time is therefore divided into intervals, and for the Berkeley UNIX 4.2BSD operating system running on VAX (*) machines this interval is 10 milliseconds.

The time-of-the-day service is important for a number of reasons:

- It provides the *real time* whenever required.
- It is the basis for *measurements* of performance and utilization of resources.
- It induces a *total ordering*¹ on the events occurring in the system.
- It is helpful in the *synchronization* of various processes.
- It produces the values of a *strictly monotonic* function

With respect to the ideal time, the behavior of a physical clock might be described in terms of the time offset, the frequency offset, and the frequency offset rate. In the case of quartz oscillators, the frequency offset can be reduced to within 1 part in 10^{11} for the thermostated ones, and the frequency offset rate is in the order of 1 part in 10^8 /day.

(*) UNIX is a trademark of Bell Laboratories.

† Work supported in part by the Defense Advanced Research Projects Agency under Contract N00039-C-0235. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

(*) VAX is a trademark of Digital Equipment Corporation.

¹ Since the time is divided into a sequence of intervals, two different events may happen during the same interval and therefore have the same timestamp. Thus, the total ordering defined above is in reality a *total quasi ordering* [Kura76].

The frequency offset, which can reasonably be thought to have normal distribution with mean zero, causes two initially synchronized clocks to drift apart and show a time offset. For example, in a VAX processor, the frequency offset of the interval clock has a maximum value of one part in 10^4 [DEC81]. This value seems higher than the normal for standard quartzes, but the manufacturer says conservatively that the accuracy may further change with temperature variations.

The frequency offset rate, the derivative of the frequency offset function, is related to the change of the characteristics of the quartz oscillator due to temperature gradients, absorption and desorption of gas, imperfection in the crystal lattice, and similar effects [Elli73], but can usually be considered zero for most standard quartzes during relatively short periods of time.

The desire for synchronized clocks in a distributed system arises from the same requirements as those described above for an accurate clock on a single machine. In our approach, we do not view the local time and the *network time* as two separate concepts, but, by identifying them, we provide several autonomous machines with a more accurate time function than those generated by their own clocks.

In the next sections we describe our algorithm and the protocol used to synchronize the various clocks. The problems that led us to the design of a new system call are briefly addressed. The results of a series of experiments that illustrate the effects of TEMPO on the network are reported. A discussion of the advantages introduced by this new service into the system is finally presented.

The Algorithm

It is possible for different machines in a loosely coupled network to compute the time offsets of their clocks. There are various algorithms [Marz83] [Elli73] for doing that. The one we devised combines simplicity and accuracy. It works as follows.

Suppose process A and process B need to evaluate the clock skew between the two different machines on which they are running. Process A (the master process) sends a timestamped message to process B. Process B (the slave process) timestamps the received message and computes:

$$d_1 = \text{timestamp}_B - \text{timestamp}_A \quad (1)$$

When a process reads the time to timestamp a message, it introduces an error (see Fig. 1) that, for simplicity of analysis, we assume to be uniformly distributed between 0 and 10 milliseconds if such is the width of the interval between two clock's ticks.²

We can therefore write:

$$d_1 = \text{time}_{B1} - e_{B1} - \text{time}_{A1} + e_{A1} \quad (2)$$

that is:

$$d_1 = (\text{time}_{B1} - \text{time}_{A1}) - (e_{B1} - e_{A1}) = T_1 + \Delta - \text{Err}_1 \quad (3)$$

where T_1 is the transmission delay, a random variable whose distribution is unknown,³ and Δ is the time offset we are trying to estimate. Err_1 may be assumed to have a triangular symmetric density function, and equals $e_{B1} - e_{A1}$.

Process B repeats the same sequence of events and sends, embedded in the message, its perception of the

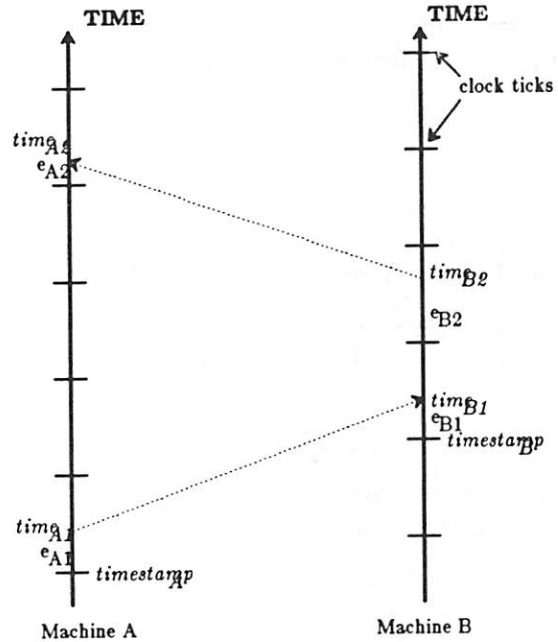


Fig.1 - Timing diagram

² Such an assumption is realistic but might be imprecise in that, due to the priority structure of the kernel, some drivers could delay the execution of the routine that updates the clock, with the result that ticks would no longer be equally spaced in time.

³ T_1 depends on several factors: the software overhead and the buffer delay on the sending machine, the network access time, the network transmission time, the software overhead and the buffer delay on the receiving machine. It may also happen that the execution of the process is not resumed immediately after the system call that reads the time returns. The expected value and the variance of T_1 are strongly system dependent: even staying within the UNIX world, the different interrupt structures of the VAX 750s and 780s, the use of different network controller boards, or the utilization of a network with one topology instead of another, may modify it.

delay time d_1 to process A.

Process A computes:

$$d_2 = (\text{time}_{A2} - \text{time}_{B2}) - (e_{A2} - e_{B2}) = T_2 - \Delta - \text{Err}_2 \quad (4)$$

and:

$$\Delta' = \frac{d_1 - d_2}{2} = \Delta + \frac{(T_1 - T_2)}{2} - \frac{(\text{Err}_1 - \text{Err}_2)}{2} \quad (5)$$

The third term on the right-hand side is a random variable whose density has the shape shown in Fig. 2; it is the convolution of the densities of Err_1 and Err_2 , since the two random variables are independent.

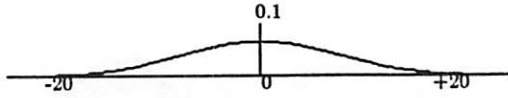


Fig.2 - The probability density function of $\text{Err}_1 - \text{Err}_2$

The second term on the right-hand side is the difference of two independent random variables with the same unknown density;⁴ hence, its density may be assumed to be symmetric.

We can now exploit these properties and compute:

$$\frac{\sum_{i=1}^N \frac{(d_{1i} - d_{2i})}{2}}{N} \quad (6)$$

where d_{1i} is computed like d_1 , and d_{2i} like d_2 .

Expression (6) can be written:⁵

$$\Delta + \frac{\sum_{i=1}^N \frac{(T_{1i} - T_{2i})}{2}}{N} - \frac{\sum_{i=1}^N \frac{(\text{Err}_{1i} - \text{Err}_{2i})}{2}}{N} \quad (7)$$

The second and the third term of (7) have mean $\mu = 0$; therefore, because of the Strong Law of Large Numbers, for N large, (6) converges to Δ .

This algorithm has been implemented and tested for $N = 16, 32$ and 64 . The details of the implementation will be described later, but here it is interesting to note that the measurements presented in all of the three cases an evident ripple, that is, a spurious oscillation around what could be supposed to be an accurate estimate of Δ . One reason for the observed behavior is that the value of expression (7) is sensitive to the variance of the

transmission delay, which happens to be very high.

Instead of further increasing N , or eliminating those values which were found to be excessively large while computing the summations in (7), a newer and simpler approach was chosen.

Suppose that we compute separately:

$$d_{1\min} = \min d_{1i}, \quad d_{2\min} = \min d_{2i} \quad (8)$$

and then:

$$\Delta'' = \frac{d_{1\min} - d_{2\min}}{2} = \Delta + \frac{\min(T_{1i} - \text{Err}_{1i}) - \min(T_{2i} - \text{Err}_{2i})}{2} \quad (9)$$

In this way, we free Δ'' from the problem of the high variance of the transmission time. In fact, the two terms to be minimized in (9) are instances of the same random variable, and the minima are two random variables with the same distribution. The variance of the minimum was much smaller than the one of the original function; hence, the difference of the two minima was found to be negligible. The above statement is based on the observation of the small variance of Δ'' .

Using the method summarized in (9), we have been able to get estimates of Δ so accurate that the sequence of data obtained varied in a strictly monotonic fashion and at a constant rate.

The software measurements revealed a rate of variation of the time offset equal to the one we obtained, using a high precision frequency counter, from hardware measurements of the clock frequencies of the machines involved in the experiment. The value of N necessary to obtain such an accuracy is surprisingly low: experiments show that, in 96% of the cases, the minimum is reached before the 7th exchanged message (see Fig. 3 and Fig. 4). These figures show the relative frequencies with which the i th message ($i=1,2,\dots$) was found to produce the minimum delay. They have been obtained by taking measurements over several days of activity.

In most cases, the first message was chosen, no matter which of the machines was acting as the master. The higher the order of a message, the fewer the times it was chosen. It is particularly interesting to observe that the histograms of the messages from Arpa (a VAX 780) to Monet (a VAX 750) are different from the ones of the messages from Monet to Arpa. This fact can be related to the different behaviors of the two machines with respect to communication facilities (especially to their different interrupt structures).

⁴ This assumption again might not be accurate if the network has different machines and communication controller boards. Furthermore, the priorities of the two communicating processes, which are recomputed by the kernel according to the amount of CPU time consumed by them, may be different, thereby affecting their response times.

⁵ It is assumed that the variations of Δ due to the relative drift of the two clocks are negligible during the time necessary to complete the measurements.

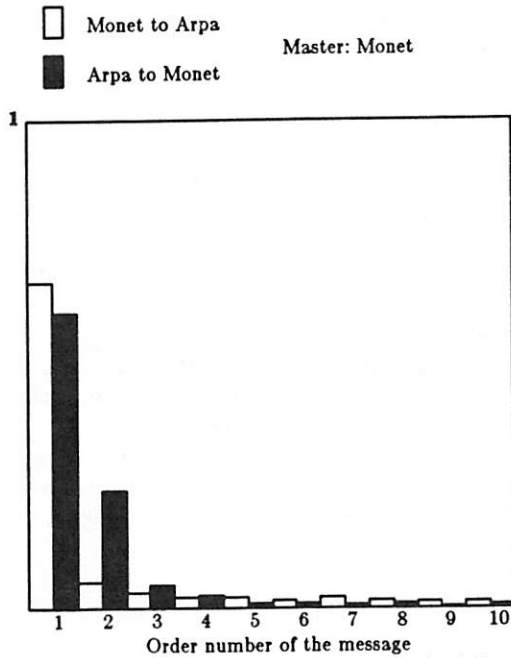


Fig.3 - Relative frequency of the selection of the i-th message as the minimum-delay message.

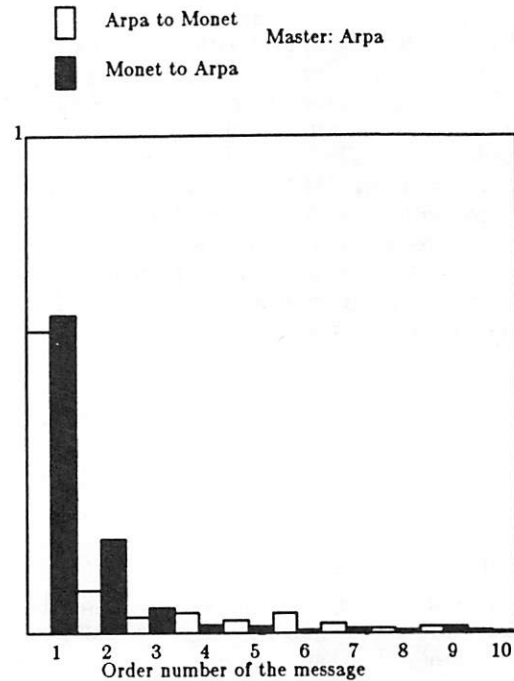


Fig.4 - Relative frequency of the selection of the i-th message as the minimum-delay message

The Underlying Communication Facility

At Berkeley, some of the machines are connected via 10 Mbit/s Ethernets, but the global communication facility is a 3 Mbit/s Ethernet. The machines are VAX 780s, VAX 750s, and SUN workstations, all running Berkeley Unix 4.2BSD. The implementation described here has been restricted so far only to the VAX's.

The basic structure for communication between machines is the *socket*. Each socket has a type that is chosen according to the communication properties visible to the user. The two most important types of communications available are *stream* and *datagram*. A *stream* socket provides a service which is bidirectional and reliable. The flow of data is sequenced and unduplicated.

Name	Type of VAX	Physical Memory	Number of Disks	Max Number of Users
Arpa	780	4M	3	32
Calder	750	2M	1	16
Dali	750	4M	3	32
Ernie	780	8M	4	64
Kim	780	4M	3	32
Matisse	750	1M	2	16
Monet	750	2M	3	16

Table 1 - The Machines of the experiments

A *datagram* socket supports a bidirectional flow of data which is not guaranteed to be sequenced, reliable, or unduplicated [Leff83]. In the Ethernet, however, messages are not duplicated or sent out of order. But, as soon as they pass through a gateway, even though only to connect to another Ethernet local area network, the order and number of messages at the destination may differ from those at the source.

The first experiments during the implementation phase of the project were performed using *stream* sockets. The results were very encouraging, but the problems caused by the delays introduced by the protocol when a message was lost and therefore retransmitted, were too cumbersome to handle in a simple way. We decided therefore to use *datagram* sockets for the

eventual implementation. Gateways were not to be crossed, and we decided not to deal with duplicated and out-of-order packets; since our software utilizes a very simple non-acknowledging protocol, taking care of these problems, though not difficult, would imply non-trivial software modifications. The advantages we obtained, using the UDP protocol [ARPA80], were a shorter transmission time with a smaller variance because of the simpler software interface and of the non-retransmission of the lost packets. In fact, our implementation of the algorithm described simply restarts the dialogue if a timeout occurs due to a lost packet.

The Protocol

Having described the basic algorithm and the communication structure on which we rely, we shall now examine the protocol that processors utilize to synchronize their clocks.

In UNIX terminology, a *daemon* is an invisible program constantly running in the background and providing some service. Slave timedemons run on all processors except on one of them, where the master timedemon runs. The master starts the synchronization mechanism, and coordinates the activities of all the other processes. The algorithm described above is implemented

by the routine *measure*, which returns the difference Δ'' between the clocks of the two machines. The basic protocol used by the master works as follows:

1. The master selects, using the Inter-Process Communication mechanism described above, one of the machines.
2. It calls *measure* and stores in an array the difference between its own clock and the clock of that machine.
3. After all the machines have been polled, it computes the *network average delta* as the average of all the different deltas.
4. It asks all the computers to correct their clocks by a quantity equal to the difference between the *network average delta* and their individual deltas.

The relative frequencies of the corrections performed on four different machines of the Berkeley Network over a period of several days are shown in Fig. 5 and Fig. 6.

The protocol does not correct the local time on a machine unless the difference between this time and the *network time* is smaller than -5 or greater than 5 milliseconds.

The precision needed determines the polling rate of the master timedemon: if for example two computers have clocks drifting apart 10s per day, to keep them synchronized within a range of 20ms we have to check them at least once every ~ 173 seconds.

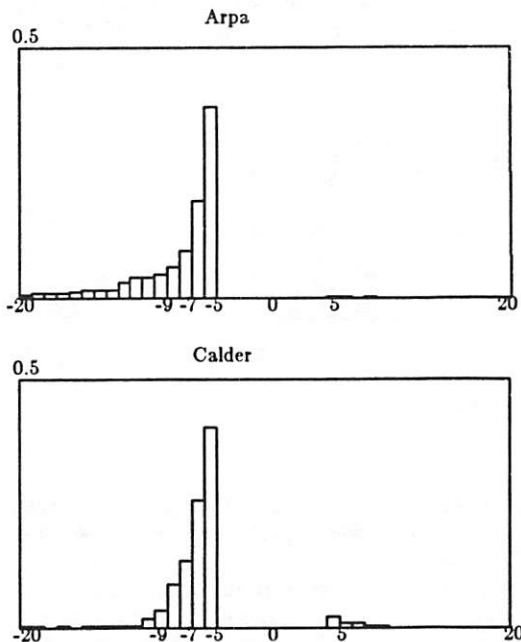


Fig.5 - Histograms of the adjustments made on Arpa and Calder

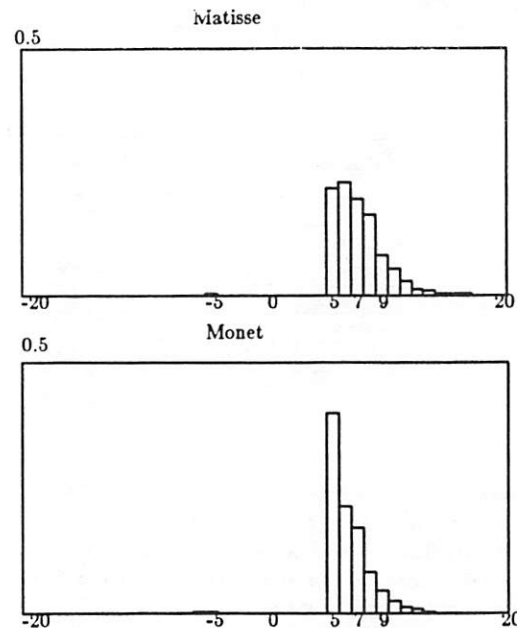


Fig.6 - Histograms of the adjustments made on Matisse and Monet

The regularity of the adjustments in our experiments was remarkable: most of the corrections were contained between 5 and 10 msec for the "slow" machines (the clocks of which tend to be left behind) and -10 and -5 msec for the "fast" ones (whose clocks tend to run ahead).

When one of the slaves does not answer the requests of the master timed daemon, after a short series of timeouts the master concludes that the host on which the slave runs has crashed, or is anyhow non-operational. When the host is reconnected to the network and the slave timed daemon is restarted, the master recognizes it and restarts polling it again with all the others. If the slaves do not receive any messages from the master within a certain amount of time, they assume the master host to be out of service, and start an election algorithm to choose another master. The algorithm we have chosen for this function is general and fault tolerant [Rica81] [LeLa77].

A new command, *tempo*, has been introduced to allow superusers to set the date on all the machines controlled by timed daemons. In fact, TEMPO will override the time changed on any single machine by the old *date* command.

Problems encountered and solutions devised

The following is a discussion of the problems we faced during the development of TEMPO.

The algorithm described above is very sensitive to the instability of any one of the machine clocks. One of our largest machines, for instance, had a clock which usually lost about two minutes per day. The network time was then affected by the "sick" machine, which induced undesirable time modifications in all of the other clocks. The countermeasure we adopted was to compute the network time as the *average* of the times of the largest set of clocks that did not differ from each other more than a predefined quantity after each set of measurements. This proved to be consistent and robust enough to take care of the problem.

Another problem was originated by a flaw of Berkeley UNIX 4.2 BSD, that until then had been harmless. The system's kernel contains a routine that is used to print various messages on the console. Sometimes, due to the special real-time requirements of some drivers in the UNIX kernel, when the print routine was called by one of those drivers running at the highest priority, the interrupts generated by the clock were ignored by the kernel for all the time necessary to print. The problem was particularly acute for one of our machines, whose console is a 300-bit/sec serial lineprinter. This virtually stopped the clock; from the point of view of the user nothing was happening, since any activities the user could see were suspended, but

from the network's point of view the timed daemons experienced long delays. This problem was solved by using an independent clock, also available on the VAX [DEC81], to check the elapsed time after each call of the routine.

Another intriguing problem we had to face was that of moving the time backward. In fact, since the *network time* is an average, some of the corrections to be made are negative in value.

An alternative approach could have been to keep track of the *differences* between the times, and provide a new system call that would correct the local time and return the network time whenever required. But in this way two different timings would have existed in the same system, and the users would have had the option (and the responsibility) of selecting the one they needed. We felt that it would be better to avoid any ambiguity by maintaining a unique network time.

In the first versions of TEMPO, the setting of the time was accomplished by means of the system calls *gettimeofday* and *settimeofday*. The sequence of operations was:

1. Get the date
2. Add or subtract the computed *delta*
3. Set the time with the new value.

Our intuition was that nobody could tell if we moved the time backwards only by a small quantity. The idea was to keep the modifications smaller than the execution time of any system calls. In this way, the time after the execution would have always been greater than the time before. For example, the order of creation of two files would have still been reflected by their creation times.

But what happened actually was that the computed delay was oscillating, and was larger than expected. Furthermore, once in a while (and so infrequently that phenomena were at first difficult to understand), the more loaded machines showed unexplainable large delays, and their clocks seemed to miss ticks. By further experimenting, we found an explanation for this strange behavior.

The timed daemons are started with the highest priority allowed to any process. Due to the particular scheduling policy of UNIX, the process priorities are recomputed once a second and changed according to the amount of CPU time obtained by the process. What happened was simply that the daemons were occasionally interrupted by the scheduler right between the systems calls *settimeofday* and *gettimeofday*: the subsequent corrections were therefore inconsistent.

The problem due to the scheduler, and the problem of setting the time backward were both solved by designing a new system call, *adjtime*, that implements the whole operation in an atomic fashion. *Adjtime* moves the time back and forward while always maintaining the monotonicity of the time function. This is accomplished by using for a suitable interval a smaller or larger increment to update the system time. So, the

value of the clock time is never decremented: its growth rate is only slowed down or accelerated depending on the situation.

The new system call allows us, however, to add to the time any quantity in only one step. We thought that this alternative choice could be useful in any cold start, when a larger correction may be required.

It has to be pointed out that TEMPO provides a "better" time than the ones of the single machines, since the inaccuracies of the respective clocks are partially corrected by the averaging operations.

A question now arises on the accuracy of the synchronization accomplished by the network time controller. It is in general very difficult to measure time discrepancies precisely; the difference in time between any two machines is a stochastic function depending on the drift of the two clocks and the corrections made by the timedemons. The following set of experiments provided us with a rough estimate of the accuracy of TEMPO.

The time necessary for a short message to go from one machine to another in either of the two directions was measured in a very large number of cases, during a whole day. The densities of the relative frequencies obtained in two of the experiments performed are shown in Fig. 7. The saw-tooth shape of these functions, which is irrelevant for the present discussion, can be explained by the statistical composition of the corrections performed by the timedemons. Every single measured value consists of the transmission time, the current delta between the two machines, and the usual error generated in the reading of the time. We were able to evaluate the range of this delta: the real difference in time varied during the day between about -10 and +30 milliseconds.

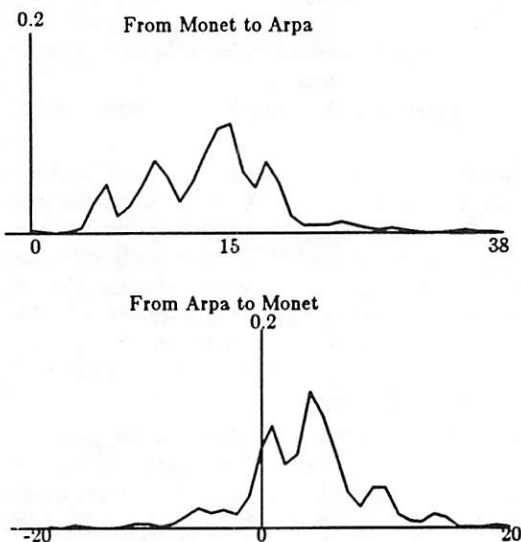


Fig. 7 - Distribution of the Measured Differences of Time between Arpa and Monet

It has to be noted however, that most of the day the time difference between the machines was within a narrower range, and that only in rare occasions it reached the extremes of the interval. In the light of these results, it can be concluded that the clocks are kept within 40 milliseconds from each other.

While in a single system we have a granularity of 10 ms to decide whether two events are coincident or consecutive, in a distributed system this granularity has a value four times as large. It seems hard to keep the clocks synchronized with a better accuracy, since the unavoidable errors range over a 40 msec interval (see Fig. 2).

The runtime requirements of the timedemons are reasonably low. The slaves, for example, consume 0.19% of the CPU time on a VAX 750, while a master running on a VAX 780 utilizes 0.1% of the CPU time per controlled machine.

Conclusions

The unique timing on which a *total ordering*⁶ of events can be based is useful to give efficient solutions to the classical problems that have been reposed by the new distributed architectures, like the mutual exclusion problem, the multiple producer-consumer problem [Rica81] [Rica79], and the scheduling of resource requests [Lamp78].

TEMPO is sufficiently reliable and fault tolerant to allow us to consider the time it keeps as a distributed system's time, common to all machines and close to the time of the real world.

One of the authors (Gusella) is working on an interprocess communication debugger, and is planning to use the ordering induced by TEMPO in the implementation of that tool.

The algorithm described above to evaluate the difference of the clocks of two machines is general enough to be implemented in other types of networks. The only condition to be fulfilled is that the distribution of transmission times can be considered the same in both directions (master to slave and slave to master). Actually, it could also work in a ring network, since the difference in the distributions of time delays due to the asymmetry in the length of the wire is negligible with respect to the other components (see footnote 3).

⁶ A total ordering can also be defined somehow arbitrarily without referring to a unique time [Lamp78], but it will not necessarily reflect the natural order of events, in the sense that an external observer could not agree with the sequence of events as seen by the system.

In the future we plan to extend TEMPO also to the SUN workstations on the Berkeley Network, in order to provide a more general service. It would be interesting to observe the performance of the time controller when operating in a very large network. The installation of TEMPO will not jeopardize the overall performance of the system, since the number of messages to be exchanged increases only linearly with the addition of new machines: the machines have, in fact, to communicate only with the master, and not with one another.

Acknowledgements

The authors would like to thank Mike Karels for his valuable suggestions and for his helpful advice on how the kernel should be modified. Bart Miller and Dave Presotto provided technical support during the implementation; their contribution goes, however, far beyond a simple work relationship. Many thanks are due to Carol Martin for her cooperation during the measurement sessions.

The authors are also grateful to the members of the PROGRES group, for their comments, ideas and suggestions; the group created a stimulating environment which facilitated the development of the ideas described in this paper.

Finally a special thank to Professor Domenico Ferrari, our Research Advisor, who never ceased to encourage us, and always found the time to help us whenever we needed assistance.

References

- [ARPA80] ARPA, "User Datagram Protocol", RFC768, *Advanced Research Projects Agency*, August 1980.
- [DEC81] DEC, VAX Hardware Handbook, 1980-81.
- [Elli73] Ellingson, C. and Kulpinski, R.J., "Dissemination of System Time", *IEEE Transactions on Communication*, no. 23, pp. 605-624, May 1973.
- [Kura76] Kuratowski, K. and Mostowski, A., *Set Theory*, North Holland, 1976.
- [Lamp78] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed Environment", *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, July 1978.
- [Leff83] Leffler, S., Fabry, R. and Joy, W., *A 4.2BSD Interprocess Communication Primer*, Computer Science Research Group, University of California, Berkeley, July 1983.
- [LeLa77] Le Lann, G., "Distributed Systems - Toward a Formal Approach", *Proceedings of the IFIP Congress 1977*, pp. 155-160, 1977.
- [Marz83] Marzullo, K.A., *Maintaining the Time in a Distributed System*, Ph.D. Thesis (DRAFT), Dept. of Comp. Sci., Stanford University, Dec. 1983.
- [Rica79] Ricart, G. and Agrawala, A.K., *Using Exact Timing to Implement Mutual Exclusion in a Distributed Network*, Tech. Report TR-742, Dept. of Comp. Sci., University of Maryland, March 1979.
- [Rica81] Ricart, G. and Agrawala, A.K., "An Optimal Algorithm for Mutual Exclusion in Computer Networks", *Communications of the ACM*, vol. 24, no.1, pp.9-17, Jan. 1981.

The version 8 network file system (abstract)

P.J. Weinberger

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Files on any of our machines may be used by arbitrary programs on any other without the program needing to know whether the file is local or remote, assuming file permissions permit.

The network file system which makes this possible is characterized by needing hardly anything from the network, minimal trust between cooperating systems, and complete system call compatibility with what existed before. Here is some salient information about the implementation, together with a little justification of the choices.

Connections are asymmetric, set up from client to host. (Clients and hosts may be quite different in size, so symmetry seemed a mistake.) The connections are virtual circuits established from a user program on the client to the server on the host. (Having the server be a user level process makes it easier to debug, and doesn't require that all kernels be changed at once.) The connection is then given to the client kernel in a new version of 'mount', so that the set of disks on the server's machine looks like a mounted file system to the client kernel. (It seemed necessary to change the kernel to provide system call compatibility. Trying to do it with a library appeared to be a maintenance nightmare.) In the kernel the routines that deal with inodes have tests to see if the inode is for a remote file system, and if so, a routine is called which makes a remote procedure call on the server using the virtual circuit.

The cooperating systems do not have to have common password files. Instead each server has a table it uses to translate between its idea of user and group, and the client's. This attacks the problem but is not a very good solution, since the set of translation tables is a distributed data base, and so difficult to maintain.

At this time the server only provides access to files, not devices, nor to further remote machines (machine a can't use machine b's connection to machine c).

Towards a Distributed File System

Walter F. Tichy

Zuwang Ruan

Department of Computer Science

Purdue University

West Lafayette, IN 47907

Net: tichy@purdue

ABSTRACT

This paper describes IBIS, a distributed file system for a network of UNIX machines. IBIS provides two levels of abstraction: file access transparency and file location transparency. File access transparency means that all files are accessed in the same way, regardless of whether they are remote or local. File location transparency hides the location of files in the network. IBIS provides a single, location transparent, hierarchical file system that spans several machines. IBIS exploits location transparency by replicating files and migrating them where they are needed. Replication and migration improve file system efficiency and fault tolerance.

This paper reports on the design, implementation, and performance of the access transparency level, and describes the design of the location transparency level.

1. Introduction

IBIS is a distributed file system for a network of UNIX machines. The purpose of IBIS is to provide a uniform, UNIX-compatible file system that spans all nodes in a network. The novel features of IBIS are file migration and replication, which improve the efficiency of file access by placing files near the network nodes where they are used. IBIS is also implemented with a new, decentralized protocol for file access. A forerunner of IBIS is STORK [1], which demonstrates the feasibility of file migration. IBIS is being built as part of the TILDE project [2], whose goal is to integrate a cluster of machines into a single, large computing engine by hiding the network.

IBIS has 2 levels of abstraction. The first level provides *file access transparency* (also called *system call transparency*). File access transparency is achieved if file access primitives like *open*, *close*, *read*, and *write* operate on any file in the network, regardless of the location of the file. Access transparency hides the access method for remote files; it does not hide the location of files.

The second level of abstraction provides *file location transparency*, which frees the user from remembering at which host a file is located. Placement of files becomes the responsibility of the file system, permitting it to exploit placement strategies that improve efficiency. *File migration* is a strategy that improves access time by placing files near the nodes where they are read and written. *File replication* is a strategy that creates copies of files to improve the efficiency of read accesses as well as the fault tolerance of the file system.

The next section describes the access transparency layer of IBIS. Besides presenting design and implementation, we discuss an authentication mechanism that guarantees the security of remote access, and provide performance measurements. Section 3 describes the design of the location transparency layer of IBIS.

2. File Access Transparency

We have extended all UNIX file access primitives to operate on both local and remote files. For example, *open* accepts a filename of the form [*<hostname>* :]*<pathname>*. If *<hostname>* is missing or the same as the current host, *open* executes a normal system call for opening the file and passes back the returned, local file descriptor. If the file is remote, *open* communicates with a server process on the remote machine to open the desired file remotely. *Open* returns a remote file descriptor in that case. All further operations on the returned file descriptor interact with the local file system or the remote server, as the case may be. Each user process, called a *client*, has its own, dedicated server process at the remote host. The dedicated server and a connection between client and server are created by a special server creation process on the remote host, at the time when the client accesses the first file on that host. If the client accesses several files on the same remote host, the dedicated server is multiplexed among all requests.

IBIS implements all remote file operations with the IPC facilities of UNIX 4.2 BSD, and is based on TCP/IP. The access transparency layer is sandwiched between the UNIX kernel and user programs. For local files, the layer simply invokes the standard UNIX system calls. For remote files, the layer observes a remote procedure call protocol for interacting with the server. Since IBIS uses TCP/IP rather than streamlined protocols, highly reliable operation results. Furthermore, IBIS is not restricted to local area networks. It can provide reliable file access between any 2 Unix systems running TCP, even if they are connected via gateways and lossy subnets. As will be shown below, the performance penalty for this generality is modest.

In std. UNIX, open file descriptors are inherited across the system calls *fork* and *execve*, which create new processes. IBIS guarantees the same behavior for remote file descriptors. For example, *fork* operates as follows (see Fig. 1). If a process with remote file descriptors forks a child process, the server forks a new server for the child on the remote machine. The child server then connects to the child process. The child server automatically shares the open files with the parent server. This protocol results in the same behavior as if the files had been local. Thus, remote file operations simulate the behavior of the UNIX file system exactly.

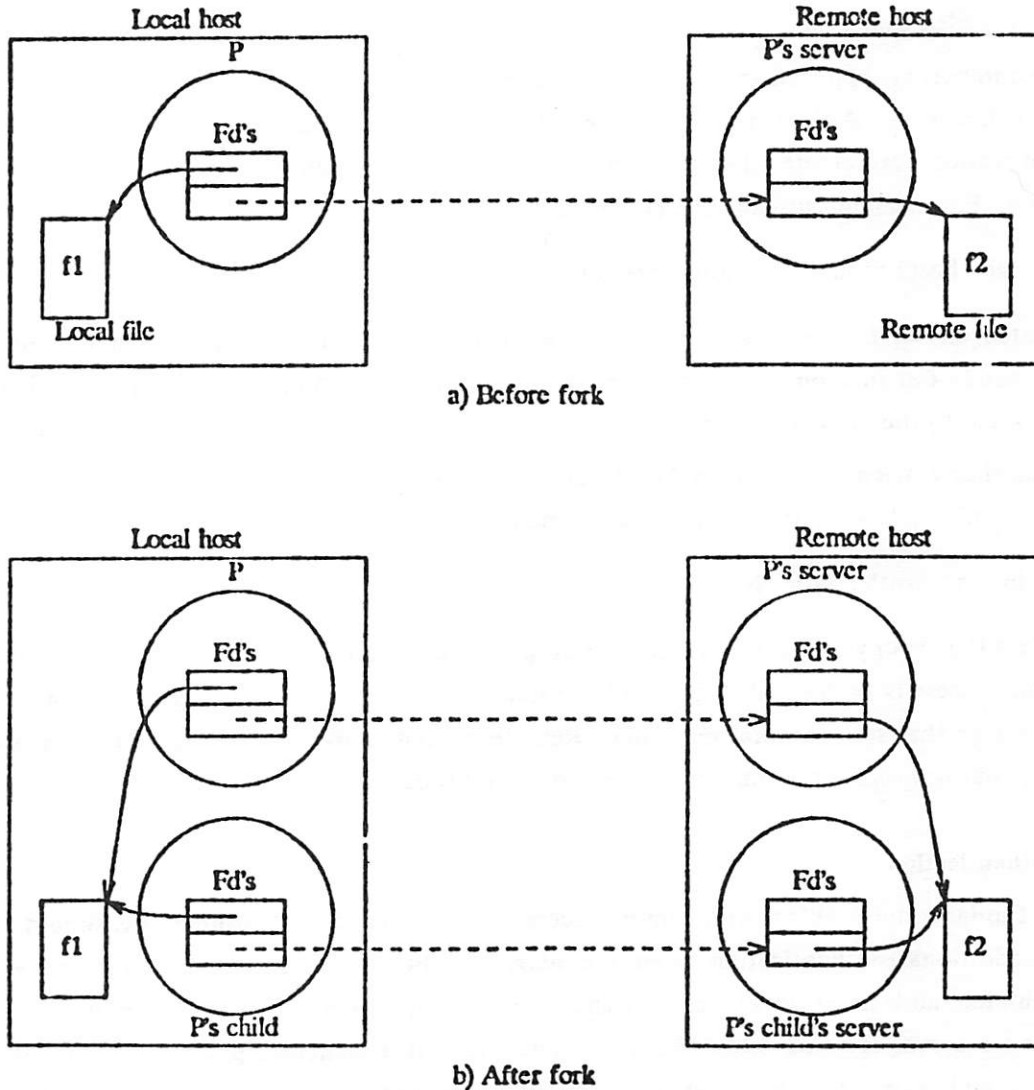


Figure 1: Remote file access before/after fork.

The complete list of system calls and I/O functions with remote access appears in the appendix. Because of faithful access transparency and a complete set of file operations, almost all existing programs can be upgraded to interact with remote files by simply relinking them with the new I/O functions. As test cases, we have relinked the following UNIX commands: *cat*, *chmod*, *cp*, *csh*, *diff*, *ed*, *ln*, *ls*, *mkdir*, *mv*, *rm*, *rmdir*, and all RCS operations [3]. For example, upgrading the command *ls* for remote access required no special treatment, since *ls* simply opens a directory as a remote file. The command *rm* required a minor modification. The Purdue version of *rm* does not remove a file; instead, it moves the file to a special directory called *tomb*, where it will be deleted after a few days. Simply relinking *rm* with the new library resulted in a program that moved a file to the *tomb* on the host where the command was executed. Thus, *rm* moved all remote files to the local machine, causing large amounts of

useless data to be transmitted. A simple modification ensured that *rm* now moves a file to the *tomb* at the file's home machine.

An important application of the remote access primitives is a version of *ssh* with remote access, called *rcsh*. Although *rcsh* executes all commands on the local machine, it provides I/O redirection for remote files, and transparent filename substitution on local and remote machines. For example, suppose the command

```
cat host2:foo.* > host3:result
```

is executed on *host1*. *Rcsh* first generates a list of filenames by performing filename substitution on *host2*. *Cat* runs on *host1*, but opens the files on *host2* remotely. *Rcsh* then redirects the output of *cat* to the file *result* on *host3*.

Another convenient shorthand made possible by the remote access primitives are cross-machine symbolic links. For example, the command

```
ln -s host1:path h1
```

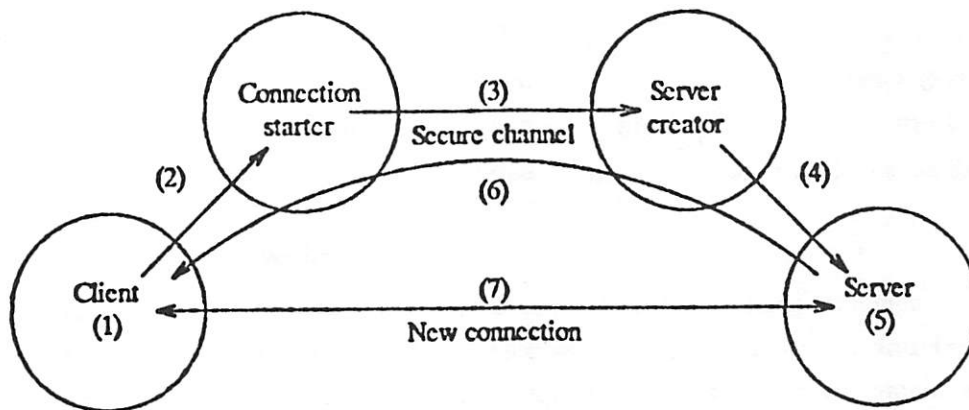
establishes the synonym *h1* for *host1:path*. Whenever a program accesses a file via *h1*, an implicit remote access is performed. With symbolic links crossing machines, a user can construct a directory tree that spans several machines. Remote symbolic links can simulate location transparency (but not replication and migration) to a certain degree.

2.1. Authentication

A fundamental problem with remote access is authentication. Remote access should not require additional authentication from the user, yet should be secure enough to prevent impersonation and violation of access rights. For example, while a connection between a client and a server is being established, it is possible that a malicious program impersonates the server and grabs the connection to the client before the real server has a chance to connect. To foil such an attack, the client needs a way of ascertaining the authenticity of the server. Similarly, the server must have a way of determining that it is connected to the right client. Finally, the server must observe the access rights that the client process has. The client process should gain no more nor less access rights through the server than if the client were running directly on the remote machine.

We solved these problems as follows. The dedicated server is established by an authentication mechanism called a "two-way handshake via secure channel" (see Fig. 2). When a client process *P* attempts to establish a connection to a remote host, it reserves a port number *A* and executes a connection starter program. The connection starter opens a secure channel to the server creator in the remote host and passes the user id of *P* to it. The server creator then establishes a server *Q* with the access rights of *P*, as indicated by the user id. *Q* reserves a portnumber *B*. Next, *P* and *Q* exchange their portnumbers via the secure channel between the connection starter and server creator. Finally, *P* and *Q* establish their own connection, and

verify each other's portnumber. This mechanism assures that neither server nor client can be impersonated by another process, and that the server has the correct access permissions. The secure channel is implemented by making both connection starter and server creator privileged processes which communicate via privileged port numbers. Note that the client and its server are not privileged and do not use privileged port numbers. A privileged channel is only used for setting up the dedicated connection. The Unix 4.2 commands *rcp* and *rsh* use the privileged channel continuously, and must therefore be privileged.



- (1) Reserve a port number A
- (2) Execute connection starter with argument A
- (3) Pass A and client's id to server creator via secure channel
- (4) Fork a server process
- (5) Reserve a port number B if authentication checking succeeds
- (6) Pass B via secure channel back to client
- (7) Establish new connection using A and B and verify

Fig.2. Two-way handshake via secure channel

2.2. Performance

We compared the execution times of our remote commands with the std. UNIX system calls for local access, and with *rcp* for remote access. Measuring execution times was difficult, since local CPU times do not reflect delays caused by communication with remote servers. We therefore chose to consider only elapsed time. Elapsed time includes delays caused by communication, but also delays caused by timesharing. The measurements were taken with only 1 or 2 users on each host, but with all other demon processes still operating. The hosts were 3 VAX/11-780s connected with a 10Mbit Pronet. All measurements are in seconds.

Table 1 below contrasts remote access and local access. Note that for local files, the overhead introduced by IBIS for *open* is negligible. For local *reads*, the overhead rises for some reason with the size of the file. Placing IBIS into the kernel would probably eliminate that anomaly. For remote files, we need to distinguish whether IBIS is accessing the first file on the remote host, or subsequent ones. The initial access is quite expensive, since it involves setting up a dedicated server and a connection. Subsequent remote *open* operations are only

about 4 times more expensive than a local *open*. The time for the initial remote *open* could be reduced by maintaining a dedicated connection per user, which is reused by every process owned by that user. A remote *read* is between 3 and 10 times more expensive than a local *read*. Again, placing the access transparency layer into the kernel should speed it up.

System call	File location	Open		Read		
		initial	non-initial	1K bytes	5K bytes	10K bytes
Standard calls	local	0.012	0.012	0.025	0.053	0.083
IBIS calls	local	0.013	0.013	0.026	0.094	0.136
IBIS calls	remote	1.787	0.053	0.078	0.377	0.884

Table 1: Performance of IBIS calls vs. standard system calls

Table 2 compares the performance of the UNIX 4.2 remote copy command (*rcp*) with the *cp* command linked with the IBIS library. The left half of the table shows copying times between a remote and the local host, the right half between 2 remote hosts. In the first case, *rcp* is about 50% slower than IBIS; in the second case about 70%. Note the file sizes chosen. According to [4] and [3], the average UNIX text file has about 250 lines, and with each lines having less than 40 characters, the average file consumes not more than 10K bytes.

Command	local <--> remote		remote <--> remote	
	10K bytes	20K bytes	10K bytes	20K bytes
IBIS cp	2.81	3.30	5.37	6.153
rcp	4.33	4.72	9.22	11.62

Table 2: Performance of IBIS cp versus rcp.

In summary, remote file access in IBIS is tolerably slower than local access. It runs somewhat faster than an existing program (*rcp*) that performs remote access. Much room for improvement remains, since IBIS was implemented with the UNIX 4.2 IPC mechanism without efficiency considerations.

3. Location Transparency, Migration, and Replication

A number of UNIX networks provide access transparency, but no location transparency, migration, or replication. Examples are COCANET [5] and UNIX United [6]. LOCUS [7] provides location transparency and replication, but no migration. LOCUS also implements a centralized file access protocol which requires that each file access must first communicate with the "synchronization site" for that file to locate a valid copy. Thus, even if a node has a

copy of a file, it must synchronize at a potentially remote site for opening and closing the file. In particular, the synchronization site of a replicated file is always remote. IBIS avoids both the bottleneck of a synchronization site as well as remote synchronization for local replicates. IBIS provides a more decentralized control for file access than LOCUS. The advantage of decentralization is more potential parallelism and better fault tolerance.

In the following, we describe IBIS' scheme for keeping a distributed and replicated file system consistent. We shall first discuss how files are treated, how replication and migration are controlled, and then present the directory level and the file lookup mechanism.

3.1. IBIS Files

An IBIS file has a unique file identifier or *fid*, which is a triple $\langle \text{host\#}, \text{device\#}, \text{inode\#} \rangle$. The *fid* of a file specifies on which host and device the file was originally created. The inode number uniquely identifies a file on a given device. Note that *fids* can be created on each host independently; no interrogation of a potentially remote synchronization site as in LOCUS is necessary.

An IBIS file is in one of 4 states: U, P, F, or S; compare Figure 3 for the complete state transition diagram. State U means that the file is a unique copy; there are no replicates anywhere. Thus, local operations on such a file incur no overhead. State P means that the file is the primary copy, and may have replicates in the network. All updates are performed on the primary copy. Replicates are cached at other sites to speed up read accesses. If the primary copy is updated, the update broadcasts a signal that invalidates the cached copies. Furthermore, if a primary copy is updated, its state reverts to U, since there are no replicates. Thus, an update has the following properties: First, update synchronizes at the file's home site; no other sites must be interrogated. Second, the cost of creating replicates is distributed to the machines with the replicates. Third, updates have the *contraction property*: They eliminate replicates and free space. Of course, the replicates will reappear if the updated file is accessed again remotely, but only at sites where the file is in use.

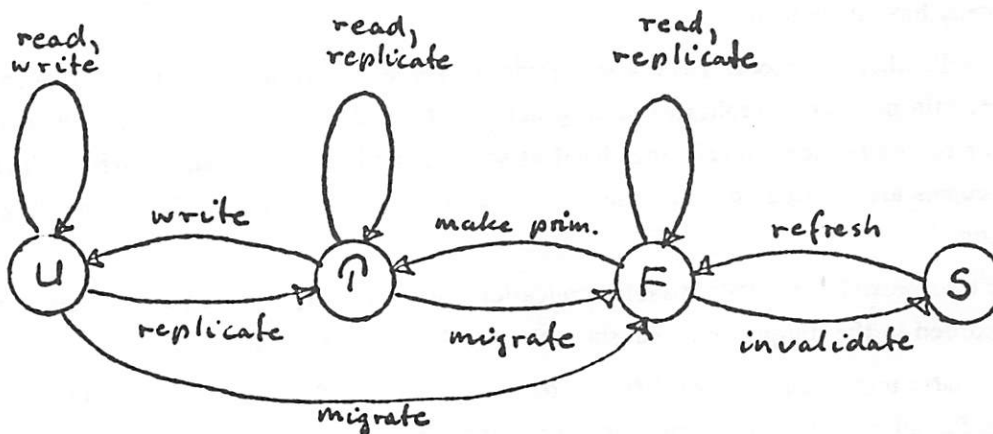


Figure 3: State transition diagram for IBIS files.

Replicated files are either in state F (fresh replicate) or S (stale replicate). State F means that no invalidation signal has been received from the primary copy, and the replicate is therefore assumed to be up to date. State S means the file is a replicate of an earlier version of the primary copy. Because of delays in propagating the invalidation signal, it is quite possible that a replicate is in state F, when it should be in state S. However, in the absence of timing channels among hosts, operations on a file are still serializable. If the delays of receiving the invalidation signal are intolerable for some applications, the best approach is to simply disallow replication of the affected files. The cost of instantaneous update of all replicates of a frequently changing file is much higher than remote access of a single copy.

Migration in IBIS means to change the site of the primary copy. Thus, migration is used mainly for speeding up write access, while replication improves read access times. Migration essentially designates a fresh copy as the new primary. It is an expensive operation, because it must atomically change the fid of the primary copy in all replicates of its parent directory (or first delete all replicates of the parent directory).

3.2. Replication and Migration Control

File replication pays off for those files that are read more often than written. Files and directories near the root of a hierarchical file system exhibit that property. In particular, directories experience much higher levels of reads than writes, and a high degree of replication undoubtedly improves system performance. IBIS therefore provides "*demand replication*" by default. Demand replication means that a replicate is cached locally whenever the corresponding primary or unique copy is read remotely. Thus, unless demand replication is disabled for a file, simply reading it generates a local copy of it. This strategy is quite appropriate near the root of the directory tree. At lower levels of the tree, sharing of sub-directories diminishes, while update traffic increases. Hence, less replication is desirable to improve performance. Note that because of the contraction property of updates, unused replicates will disappear after the next update. We therefore expect lower levels of the tree to automatically have little or no replication.

For reliability purposes, IBIS also provides "*forced replication*". Forced replication causes a certain number of replicates to be generated, even if no remote access occurs. A copy marked for forced replication refreshes itself as soon as the invalidation signal arrives. (Thus, replicate copies are "pulled" by the remote site, rather than "pushed" by the site holding the primary copy.)

It is also possible to totally disable replication for frequently updated files. This restriction is recorded in the unique copy, causing the state U to "stick" to it.

Automatic migration is more difficult to implement. IBIS will initially provide explicit commands for migration. As we gain more experience with network operating systems where processes seek out the hosts with the lowest loads, we plan to develop automatic placement and migration mechanisms.

3.3. Directories and Pathname Searching

Directories are implemented as files. They have the same state attributes and follow the same access protocol. A directory simply pairs character strings with fids of files (which may again be directories). Because the fid may belong to a remote file, and because there may be a local replicate, the directory must contain additional information, namely the fid of the local copy (if any). Thus, a directory implements the following 2 mappings:

character string --> primary fid --> local fid

If the primary copy is on the local host, the primary fid and the local fid are identical. The local fid is undefined if there exists no local copy. A directory entry with an undefined local fid is called a "dead end". Whenever directory lookup reaches a dead end, remote access is needed for locating the object. Fig. 4 below illustrates two hosts with two levels of a common directory tree which is partially replicated.

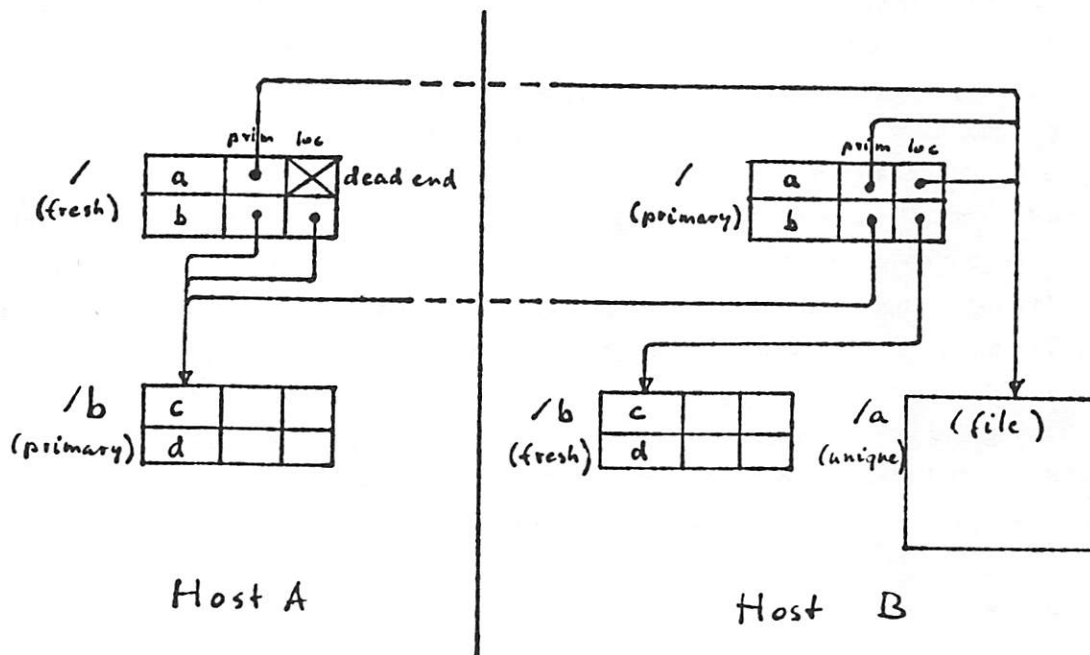


Figure 4: A replicated directory tree.

Replicated copies of directories are in general NOT identical, even if they are consistent. For instance, two directories on two machines listing a replicated file contain different local fids. When a stale directory replicate is refreshed, a simple copy operation is not sufficient. What must be updated is the mapping from character string to primary fid, but the mapping of primary fid to local fid must remain unchanged (except for deletions). Otherwise, the replicate of a whole subtree may be lost whenever the root of that subtree changes. For example, in Figure 4 consider what must be done in host A if the entry b in the root directory of host B is renamed to x.

It should now be clear how pathname searches proceed. Given a character string, a pathname search locates the primary fid and the local fid (if defined) of the file named by the character string. The local fid is desirable for local read access; the primary fid is needed for (potentially remote) read/write access. The search starts either with the current directory, or the file system root. To commence, one of these two directories is opened. This may be a local open of a unique or primary copy, or a remote open, or an open of a local replicate (possibly after restoring its state to F). The opened directory is searched for the first pathname component. If a match is found, the associated primary and local fids are retrieved. If the local fid exists, the search continues in the replicate. If the entry is a dead end, the search must continue in a remote directory. The host number embedded in the primary fid indicates where the primary is located. If the host with the primary is not reachable because of network partitioning, a broadcast may locate a replicate. After the primary or a replicate is located, the search continues in that directory (possibly creating a local replicate at the same time).

Much design work remains to be done. We need to develop robust algorithms for partitioning the net and the file system in case of failures, and for reconnecting it back together. If the network is partitioned, primary copies of files may no longer be reachable. In that case, a special protocol must designate one of the (hopefully available) replicates as a temporary primary, such that updates may proceed. For reconnecting the file system, we have already developed a general three-way file merging technique that can merge two versions of a file with respect to a common ancestor [8]. A three-way file merge is reliable, provided there are no overlapping changes and the common ancestor of the two diverging primaries has been saved. The natural time to save the common ancestor is when a replicate is promoted to temporary primary status. Our merging technique merely requires that the user provide routines for extracting and comparing individual records for each type of file to be merged. Merging of directories and text files is done automatically. The merging technique generalizes to a $3(n-1)$ -way merge, if the net is partitioned into $n \geq 2$ subnets.

4. Conclusions

The access transparency layer of IBIS is complete and operational. Using it is immediately addictive. After some initial experimentation, it becomes natural to access remote files, especially since there is not a single new command to learn. Directly editing a remote file is enormously more convenient than using remote login. Interactive programs with remote access (like screen editors) are normally faster and need fewer cycles than remote login, because all high-bandwidth user interaction is done locally. Moreover, remote access commands can deal with files on several machines simultaneously, whereas remote login or a remote shell limits the user to one machine at a time. Finally, any existing program can be upgraded to remote access almost instantly.

Remote symbolic links have also proven to be quite useful. They permit users to simulate a directory tree spanning several machines. Remote symbolic links in conjunction with access transparency can almost provide location transparency. The only drawback is that the

current directory must be on the local machine; it is impossible to change the current directory to a remote one. We plan to lift this restriction.

The access protocol for the location transparent level of IBIS has been carefully designed to avoid overhead for purely local operations. For example, local file creation, local read/write of a unique or primary copy, and local read of a replicate incur almost no overhead. In addition, the cost of making replicates is distributed, and the contraction property of updating assures that frequently written files experience a low level of replication. Demand replication, on the other hand, automatically replicates files and directories that are read-shared at many different hosts. File migration, finally, improves write access by automatically or semi-automatically moving files to the sites where they are written.

Appendix: The IBIS Access Transparency Layer

System calls with remote access:

access, chdir, chmod, close, creat, dup, dup2, fchmod, flock, fork, fstat, fsync, ftruncate, link, lseek, lstat, mkdir, open, read, readlink, rename, rmdir, stat, symlink, truncate, umask, unlink, write.

Stdio function with remote access:

clearerr, fclose, feof, ferror, fflush, fgetc, fgets, fileno, fprintf, fputc, fputs, fread, fscanf, fseek, ftell, fwrite,getc, getchar, gets, getw, printf, putc, putchar, puts, putw, rewind, scanf, setbuf, setbuffer, setlinebuf, sprintf, sscanf, ungetc.

Other library functions with remote access:

closedir, opendir, perror, popen, readdir, rewinddir, scandir, seekdir, telldir.

References

1. Paris, Jehan Francois and Tichy, Walter F., "STORK: An Experimental Migrating File System For Computer Networks," pp. 168-175 in *Proceedings IEEE INFOCOM*, IEEE Computer Society Press (April 1983).
2. Comer, Douglas, "Transparent Integrated Local and Distributed Environment (TILDE) Project Overview," CSD-TR-466, Technical Report, Purdue University, Department of Computer Science (1984).
3. Tichy, Walter F., "Design, Implementation, and Evaluation of a Revision Control System," pp. 58-67 in *Proceedings of the 6th International Conference on Software Engineering*, IPS, ACM, IEEE, NBS (September 1982).
4. Kernighan, Brian W. and Mashey, John R., "The UNIX Programming Environment," *Software -- Practice and Experience* 9(1) p. 1-15 (Jan. 1979).
5. Rowe, Lawrence A., Cortopassi, Joseph R., Doucette, Douglas P., and Shoens, Kurt A., *RIGEL Language Specification*, Computer Science Division, University of California, Berkeley (March 1980).
6. Brownbridge, D. R., Marshall, L. F., and Randell, B., "The Newcastle Connection or UNIXes of the World Unite!," *Software -- Practice and Experience* 12 p. 1147-1162 (1982).
7. Popek, Gerald, Walker, Bruce, English, Robert, Kline, Charles, and Thiel, Greg, "The LOCUS Distributed Operating System," *Operating Systems Review* 17(5) p. 49-70 ACM, (Oct. 1983).
8. Tichy, Walter F., *The String-to-String Correction Problem with Block Moves*, to appear in ACM TOCS 1984.

The Livermore Interactive Network Communication System

Alex Phillips

Lawrence Livermore National Laboratory

Abstract

The Livermore Interactive Network Communication System (LINCS), is a protocol suite that supports a location independent application environment. The Livermore Computing Center at the Lawrence Livermore National Laboratory has a large heterogeneous network. A network operating system NLTSS is being developed on this network. LINCS provides not only the communication services necessary to support location transparent applications but also provides translation for machines of varying word lengths and data representation.

This paper will describe the protocols that comprise LINCS. The ISO- OSI network model will be used as a point of reference. In the reference model, LINCS occupies the application, presentation, session, transport, network, and link layers.

The application layer has customers and servers. A server is a program that provides a service, and a customer is a program that needs some service. A server may need other services from other servers, in which case the server becomes a customer of another server. Customers and servers communicate through ports. A port is simply a network address. A pair of addresses form an association through which two processes, customer and server, can communicate.

The presentation layer takes requests from a customer in the form of two data structures, a monologue control record and a token map and transforms the request into a machine independent bit stream called a token stream. The server's presentation layer interprets the token stream and fills the server's monologue control record according to the specifications of the token map. The server would then usually issue a reply via the presentation layer in the same fashion.

The session layer places synchronization marks on the token stream. There is a begin (B) mark, and end (E) mark and a wakeup (W) mark. Requests begin with a B and end with an E. This prevents partial requests or replies from being acted on. The W mark

wakes a process at a point where servicing can begin.

The transport layer consists of the connectionless timer based transport protocol Delta-t. Delta-t requires no open and close negotiation or hand shake. It provides a reliable, flow controlled channel between two ports. No bits are lost, damaged, duplicated or missequenced. When the channel has not been used for some time, the channels state reverts to a default state, and all connection records are reclaimed automatically.

The network layer routes packets produced and consumed by the transport layer. It also checks for damaged packet headers and enforces packet lifetimes. Timer based protocols must have strictly enforced packet lifetimes to operate correctly.

The link layer protocol is ALP. A simple sliding window, sequenced packet protocol. It provides more rapid retries than the transport layer could, in the event of damaged or lost packets.

LINCS has been implemented on the Cray-XMP, Cray-1s, VAX-Unix 4.2bsd, VAX-VMS, and SEL32. It is being implemented on an IBM-PC, LSI-11, SUN workstation, and CYB68000. The communication mediums include 50-Mbit NSC Hyperchannel, Ungermann-Bass Net-One broadband, 10-Mbit Ethernet and synchronous and asynchronous lines ranging in speed from 1/4Mbit to 9600 baud.

An Optimizing Portable C Compiler for the New CDC CYBER 180

Kok-Weng Lee
Mario D. Ruggiero

Human Computing Resources Corporation
Toronto, Ontario, Canada

ABSTRACT

The CYBER 180 is a recently developed 64-bit byte-addressable computer manufactured by the Control Data Corporation; it has many unique architectural features, including a massive address space, a dynamic module linking mechanism, a rich set of registers, and a ring protection mechanism. This paper presents the design decisions taken in the implementation of a C compiler for the CDC CYBER 180 computer. To put these design issues in perspective, an overview of the CYBER 180 architecture is also given.

1. Introduction

The C compiler developed for the CYBER 180* is an extensively modified version of the portable C compiler provided by AT&T as part of the System V UNIX** system for the VAX***. It implements a complete, correct and

standard version of C. Any program that follows the definition of C in The C Programming Language by Kernighan and Ritchie [Kern78] will be correctly compiled. The CYBER 180 compiler was originally developed as a cross-compiler running under System V VAX UNIX. It is intended to run both in a UNIX environment and on the native operating system provided by CYBER 180. The desire to maintain compatibility with existing C compilers and thus ensure portability of existing UNIX programs was an important factor in the design; we will highlight some of the issues

* CYBER is a Trademark of Control Data Corporation.

** UNIX is a Trademark of AT&T Bell Laboratories.

*** VAX and PDP are Trademarks of Digital Equipment Corporation.

in Section 3. Cross-compiling caused some problems due to architectural differences between the VAX and the CYBER 180. Some of these will be described in Section 4.

One of the main goals of the implementation was to produce a compiler which generates highly optimized code. Since there are a large number of registers available on the CYBER 180, the compiler optimizes usage of local variables by keeping them in registers as long as possible. The C compiler also uses a modified version of the VAX c2 optimizer to do some additional peephole and branching optimization. We will discuss the optimization issues in Section 6.

2. Introduction to the CYBER 180

This section presents an overview of the CYBER 180 architecture. The features which had a significant impact on the design and implementation of the C compiler are given particular emphasis.

2.1. CPU Architecture

The central processor contains 16 64-bit general purpose data registers (called X registers); these are used to hold logical quantities, signed integers and signed floating-point numbers. The processor also contains 16 48-bit address (or A) registers which are used to hold the addresses of operands in central memory. The Program Address (or P) register is a 64-bit register which contains the address of an instruction in central memory during the time it is read, interpreted and executed by the processor. It also carries information used by the protection mechanism.

The CYBER 180 instructions are for the most part 2-operand instructions. The first operand of an instruction is almost always a register. The second operand may be a register or a base register plus displacement (the displacement ranges from 2^{12} to 2^{19} bytes depending on the instruction). Some instructions allow an index register to be specified in conjunction with the base and displacement. The Business Data Processing instructions use a memory-to-memory structure; each operand is specified by a data descriptor containing the base and displacement plus the length and type of the operand.

2.2. Virtual Memory

The basic element of the security and protection mechanism of the CYBER 180 is the memory segment. Each task on the CYBER 180 executes in its own unique address space which consists of one or more segments. Each segment may contain up to 2^{31} bytes. The Segment Descriptor Entry (SDE) for a given segment contains information such as the size of the segment and access attributes used to enforce the protection mechanism. Segments are divided into pages; the operating system performs management of the virtual memory system at the page level. The size of a page is software selectable in the range 512 bytes to 64 Kbytes.

All addresses generated by programs are Process Virtual Addresses (PVAs). A PVA is 48 bits wide; it contains a 4-bit ring number, a 12-bit segment number, and 32-bit byte offset within the segment. The high-order bit of the byte offset is used only as a guard bit to prevent address arithmetic from

crossing segment boundaries; thus the actual byte offset is only 31 bits. The PVA is converted to a Real Memory Address in the conventional way through a per-process Segment Descriptor Table and a system-wide System Page Table. The hardware provides and manages a number of caches of recently used segment and page table entries to reduce the number of memory references used in decoding a PVA.

2.3. Protection Rings

In order to control access to memory, each address space on the CYBER 180 is organized into 15 Rings of Protection. They may be regarded as separate machine states having differing privileges. The rings are organized hierarchically so that the lower the ring number the higher the privilege; ring 1 has the highest privilege. In general, code residing in ring n can read and write segments in ring n and in higher numbered rings. In addition, code in ring n can call procedures in ring n and in lower numbered rings, although such calling is carefully controlled. The SDE for each segment contains several ring numbers; these define four ring brackets which control access for read, write, execute, and call.

It is common for a procedure to perform work on behalf of another, less privileged, procedure and it is important to ensure that the more privileged procedure does not act with greater authority than has been assigned to the caller. To this end, whenever an A register is loaded, either explicitly (via a ``load'' or ``copy'' instruction), or implicitly (via a ``return'' or ``pop'' instruction) the hardware makes a

comparison between the ring number of the A register being used to point to the source data for the load, the ring number of the data being loaded itself, and one of the ring numbers of the SDE associated with the source data. The ring number corresponding to the least privilege is placed into the destination A register.

2.4. Binding Segment

All program text is inherently sharable in the CYBER 180, which means that each executing task must have a way of uniquely identifying its data. This is done by forcing all references to global data items to be made indirectly through the binding segment. The loader places the address of the required data item in the binding segment; the address of the binding segment is passed to a shared routine when it is called. In this way a single copy of a given routine can be used by multiple processes.

A second use for the binding segment is to control access to non-local subroutines. This is accomplished by calling non-local subroutines indirectly through the binding segment. Again, the binding segment contains the address of the routine to be executed. The entry also contains a ring field which is used to validate calls to the routine; in addition, for calls to external routines, it contains a pointer to the binding section to be passed to the called routine.

2.5. Dynamic Linking Support

Ring number zero has been reserved for a special purpose, namely Dynamic Linking. On the CYBER 180, a procedure does not

need to be linked into a program before execution starts - it can be linked into the program the first time it is called. In this way, if a procedure is never called in the execution of a program, it need never go through the linking mechanism, and will never require that memory be allocated to it. A ring number of zero is reserved to denote an unlinked pointer to such a routine.

When a ring number zero is detected on a ``call'' instruction, the CYBER 180 will cause an interrupt to be taken which will give the operating system an opportunity to link the required module into the executing program before transferring control to it. Because the contents of an A register can also be used to branch indirectly to a section of code, it is also necessary to detect the loading of a zero ring number into an A register. This causes a trap sequence similar to that for a call using a pointer with a ring number of zero. This feature has a significant impact on the design and implementation of the compiler.

3. Design Issues

In this section, we present the design decisions taken in the implementation of the C compiler and try to explain the reasons behind each decision.

3.1. Data Type Issues

A number of the design issues are related to the representation of C data types.

3.1.1. Pointer Representation

Pointers are 48-bit quantities in the CYBER 180, while integers are 64 bits wide.

Although the definition of C does not specify the size of a pointer, there are many existing C programs which use pointers and integers interchangeably. In order to ensure that these programs execute correctly on the CYBER 180, we decided to artificially increase the size of pointers so that they are the same size as integers. Therefore, a pointer occupies 8 bytes of memory and is right-aligned in a word.

3.1.2. Floating-Point Numbers

We have decided to use the same representation for both single and double precision floating-point quantities: both use the 8-byte CYBER 180 single precision floating-point format. This format consists of a sign, a 15-bit biased exponent, and a 48-bit fraction. Although there is a full complement of arithmetic instructions which operate on double precision floating-point numbers, they are slower than the single precision floating-point instructions. Moreover, there are no instructions to directly compare two double precision floating-point numbers*.

Since the definition of C specifies that ``all floating arithmetic in C is carried out in double precision'' [Kern78], all programs which perform floating-point operations will be penalized if we implement double precision floating-point with the

* Of course, the comparison could be performed by subtracting the two numbers (using a subtract double instruction) and comparing the high-order part of the result to 0.

128-bit format. We believe this is unacceptable since the single precision format will meet the requirements for accuracy in most applications and these applications should not be forced to suffer in performance. Therefore, we decided to use the 64-bit format for both single and double precision floating-point numbers. The full set of double precision operations is available at the assembler level, and these could be used to implement a package of double precision routines which could be called from C programs.

3.2. More Pointer Issues

There are a number of design issues related to the operations defined on pointers. The design decisions in this area were constrained by the architecture of the CYBER 180.

3.2.1. The NULL Pointer

The value zero is widely used as the value of a NULL pointer in C programs. As explained in Section 2, the protection mechanism reserves a zero ring number to indicate dynamic linking and causes a trap. Thus a value of zero is not the best choice for the NULL pointer (CYBIL, the CYBER 180 implementation language, actually uses 0xffff80000000). Explicit assignment of zero to a pointer could be discovered by the compiler and changed to use a value that is more natural for the hardware. However, many existing C programs depend on implicit assignment of zero to pointers by virtue of the fact that memory is initialized to zero when a program begins execution in UNIX, and hence do not explicitly initialize pointers to any value when NULL is required. The

compiler must ensure that the zero value can be used as the NULL pointer.

In general this involves loading the desired pointer value into an X register and then copying the X register to an A register. Under these circumstances the hardware does not cause a trap when the X register containing the zero ring number is copied to the A register. (Note that the ring number comparison described earlier is still performed, and the resulting A register will have the ring number set to something other than zero - the only difference is that the trap is not taken.)

3.2.2. Ring Number in Pointers

The ring number inserted by the protection mechanism into operands that are placed in A registers can cause problems in expressions containing pointers. The compiler attempts to ensure that expressions are evaluated in a way that guarantees that the ring numbers are inserted in the same way in all the operands. Nonetheless, it is possible to write C programs which will not work as expected because of the insertion of the ring number. For example, if *p* is defined and used as follows:

```
char *p = (char *) 0;

if ( (int) p == 0 )
    . . .
if ( p == (char *) 0 )
    . . .
```

the first ``if'' statement will fail because *p* will have the ring number set, whereas the second will succeed. A solution to this problem would be to mask the ring number field in a pointer whenever it is cast to an integer.

The current version of the compiler does not do this, however.

3.2.3. Pointer Alignment

Since a pointer is a 6-byte quantity occupying an 8-byte word, we had to make a decision about right or left justifying the pointer. To allow existing C programs which may use pointers and integers interchangeably to work properly, pointers must be right justified. However, the CYBIL language for the CYBER 180 uses left justified pointers when passing VAR-style arguments to procedures, so our choice causes difficulty in interfacing to procedures written in CYBIL. Fortunately, C allows considerable flexibility in the manipulation of data values, so that we have been able to create the equivalent of left-justified pointers for these special cases by explicit manipulation. Nonetheless, the responsibility for the correctness of this approach lies with the programmer.

3.3. Dynamic Linking Issues

The dynamic linking features complicate the subroutine call mechanism as all non-local calls must be made indirectly through the binding segment. This segment is maintained by the loader and holds the addresses of all globally accessible symbols (i.e. text and data). The compiler outputs information used by the loader to create the binding segment. Storage of pointers to functions is also affected: the address of a function itself cannot be stored or passed as an argument. Rather, the address of the entry in the binding segment describing the function must be used.

Similar problems arise with respect to the addressing of global data items. Since any global item in one compilation unit can potentially be accessed by any other (separately compiled) unit, global data must also be accessed through the binding segment.

A problem with the binding segment approach is that the instruction formats limit the size of the binding segment that can be accessed in one "load address" instruction to 64 Kbytes. Since binding segment entries occupy either 8 or 16 bytes each, this limits the number of entries to 4096 in the worst case. This represents an upper bound on the total number of procedures, global variables, and static variables (which are treated as globals by the compiler but are not allowed to be referenced from outside the procedure block which declares them). This limit is probably adequate in most cases, but there is some fear of overflow of the binding segment.

The scheme used by the current compiler for access to the binding segment limits it to 64 Kbytes. This could be modified to increase the size of the binding segment that can be accessed. One way this could be done is to load the required entry first into an X register and then copy it to an A register. This would increase the maximum size of the binding segment to 512 Kbytes, and the worst case limit on the number of entries to 32768.

3.4. Function Arguments

The C Standard does not specify the order of evaluation of arguments to functions; the earliest C compilers pushed the

arguments onto the stack from right to left (i.e. the last argument was pushed first). Because the stack on the CYBER 180 grows towards higher addresses, we chose to reverse the order of stacking the arguments; in the part of the stack used for the arguments, the first argument occupies the lowest memory address and later arguments occupy higher addresses.

This scheme has a number of advantages. First, the layout of arguments on the stack is the same as for the VAX, so that it will be possible to easily port even programs which make assumptions about the memory layout on the VAX. Furthermore, this scheme is compatible with the argument order in CYBIL and makes it possible to call CYBIL routines (although the problems of pointer alignment remain). On the other hand, this method can cause unexpected results if argument expressions have side effects.

3.5. Function Return Values

Values are returned by functions in one X register. This is not appropriate for a pointer value which should be returned in an address register. All values must be returned in an X register, however, because it is not always possible in C to tell if a (called) routine is returning an integer or a pointer.

4. Problems of Cross-compiling

There are a number of issues related to cross-compiling which are not handled very well in the standard System V C compiler and assembler that we used as the base for the CYBER 180 C compiler. This is not unexpected, in that these programs were not

meant to be used as cross-compilation tools. Nonetheless, these problems represent oversights in the design of the programs; a number of modifications to these programs would have made them more useful as starting points for the compiler effort.

A significant difference between the architecture of the VAX and that of the CYBER 180 is the byte order within a word. In the VAX the low-order byte of a word has the lowest address, whereas on the CYBER 180 the low-order byte of a word has the highest address. This becomes an issue in the assembler when actual bytes of code are to be emitted. A small change was required to the output routines to cause the bytes to be output in the right order. It is interesting to note that this code can be written so that the output is correct regardless of the host machine on which the assembler is running. This is important in order to avoid the proliferation of compile-time options which must be specified if this type of problem is handled with conditional compilation. Furthermore, conditional compilation would result in a family of assemblers for different environments. Each of these assemblers would have to be tested in its own environment, and the complexity of these tests increases with the number of different versions possible.

The handling of floating-point constants presents a similar problem. Again, the assembler routines which create the bit representation for floating-point numbers were modified to reflect the representation of these numbers on the target machine. This can also be done in a machine-independent way so

that the same routine will work when run on the target machine or when cross-compiling.

A somewhat different problem is posed by the initialization of memory words. This stems from another difference in the architectures of the VAX and the CYBER 180, namely the size of a memory word. Basically, the compiler and assembler maintain internal representations for constants they encounter in terms of the int data type, which maps into a 4-byte word on the VAX and into an 8-byte word on the CYBER 180. When cross-compiling a program which attempts to create an 8-byte constant, the compiler and assembler will limit the value to the 4-byte structure they have available internally. This represents a fundamental flaw in these programs, as far as cross-compiling is concerned. The solution is to allow all values to be represented by an array of a basic data type (e.g. byte) which will be available in all the environments where the compiler will be required to run. The size of this array can be made a parameter of the implementation. This approach requires that all operations on these internal structures be performed by explicit calls to subroutines which perform operations such as add, subtract, compare, etc. This represents some penalty in performance when the program is not run as a cross-compiler, because a more appropriate data structure is available. Much more important, however, is the fact that it removes a significant (but implicit) machine dependence from these programs.

In our implementation, we could not afford the time to make such a large modification to the basic data structures in the

compiler and assembler. Consequently we settled for ensuring that sign extension is done properly and forcing programs which need large constants to use variables which are defined (at the C level) to be structures consisting of data types which can be handled by the cross-compiler and initialized explicitly by their component parts. This approach was feasible only because much of the code which needs to be ported does not use large constants. It is very tedious and somewhat error-prone.

5. New Features

5.1. Register Floating-Point

We have implemented floating-point register variables in the C compiler. Traditionally, register declaration only applied to variables of type int, char and to pointers; automatic floating-point variables (single and double precision) were always kept on the stack. The historical reasons for these restrictions could be due to the fact that PDP-11 has a relatively small number of registers and all floating-point operations have to be done in double precision (which would require two registers on the PDP-11). Since all floating-point variables on the CYBER 180 fit into a single X register and there is a large set of registers available, we have decided to allow register declaration for floating-point variables; in this way, the performance of programs which perform substantial floating-point operations can be improved.

5.2. Local Stack Variables

Historically, only the first few register declarations in a function are effective (three on

the PDP-11, more on the VAX). In order to make full use of the rich register set on the CYBER 180, we have allowed as many as ten data registers and eight address registers to be used to implement register variables.

Although the restriction of only three register variables was removed from our C compiler, many existing C programs have not been written to take advantage of this feature. In order to improve these existing programs without having to modify them, our C compiler will automatically keep automatic scalar variables (excluding arguments) in registers if it is possible. To achieve this, the compiler first allocates registers for automatic register variables declared explicitly by the user; then for each local automatic (scalar) variable, the compiler will try to keep it in a register if there is a register of the correct type (address or data) available and if no address operator '&' has been applied to this variable. This feature has proved to be very useful and has improved some of the existing programs substantially both in execution speed and program size.

6. A powerful Back End Optimizer

We have developed a 'back end' optimizer which will take the assembly code output by the C compiler and apply a number of transformations designed to improve speed and decrease code size. The optimizer is derived from the standard System V 'portable code improver' (PCI) that was enhanced by HCR. Like the portable C compiler, PCI is organized as a machine dependent part and a machine independent part.

The machine independent part implements the following optimization techniques:

- 1) Removal of unreferenced labels
- 2) Branch chain shortening
- 3) Deletion of unreachable code
- 4) Merging of common code
- 5) Code re-ordering and loop rotation
- 6) Redundant load/store elimination

In addition, this part of PCI provides code implementing live/dead register analysis.

The machine dependent part consists of:

- 1) A LEX script that analyzes the assembly language input file and creates the data structures used by the machine independent portion.
- 2) A collection of routines used to analyze assembly language operands.
- 3) The peephole optimization routines.

7. Future Plans on Optimization

7.1. Function Arguments

We are considering keeping function arguments in registers automatically, in much the same way as local variables are assigned to registers. In this case however, there is an added consideration. It is possible for a procedure to obtain the address of one argument and to use pointer arithmetic to access other arguments. This is typically done using the 'varargs' macros; a typical example of a program which does this is `printf`, which takes the address of the second argument and then accesses the rest of the arguments by moving the pointer. If

arguments are to be assigned to registers, then the compiler must check whether the address operator ``&'' is used on any of the arguments. If it is, then none of the arguments can be assigned to registers, because they may be referenced indirectly through this pointer.

7.2. Small Structures

Currently, the C compiler returns a structure by actually returning a pointer to the structure; the structure data is then copied by the calling program. This is not very efficient for structures of 8 bytes or less. We plan to return small structures in the returning register directly.

8. Conclusions

Despite the fact that the CYBER 180 was not designed with the C language in mind, it has proven relatively easy to construct a C compiler for this machine. Of the problems encountered, many have been due to the protection mechanism provided by the CYBER 180. This is not surprising since it is in this area that the hardware is most unlike the machines that have traditionally been used for C. The design problems dealing with the implementation of pointers are also related to the architecture of the CYBER 180; in this case they are compounded by the need to maintain compatibility with old software which was much more machine-dependent.

Acknowledgements

We would like to acknowledge the contributions made to this project by other members of the technical staff at HCR: Paul Neelands and Allen McIntosh made the

original enhancements to PCI; Teresa Wong was responsible for much of the testing of the CYBER 180 C compiler; technical management for the project was under the direction of Sheila Crossey and Richard Sniderman.

References

Kern78 B.W. Kernighan and D.M. Ritchie, The C Programming Language, Prentice-Hall, 1978.

A Simple Simulation Toolkit in C

Robert P. Warnock, III

Fortune Systems Corporation
Redwood City, California 94061

Bakul Shah

458 Ferne
Palo Alto, CA 94306

ABSTRACT

A simple set of simulation tools is presented which offers much of the power of specialized simulation languages such as GPSS or SIMSCRIPT, while not interfering with the C programmer's native style. The toolkit, a library of C routines, comprises three conceptual layers (the user's simulation is the fourth), each reflecting some well-known design borrowed from other systems. From the "bottom" up, they are: (1) Co-routines and process objects; (2) Condition variables and priority queues; and, (3) A simulation kernel (the "clock" object and the "busy" statement). Co-routines and process objects are as in the BLISS language. Condition variables, priority queues, and the simulation kernel itself are from Holt's Concurrent Euclid. The overall idea of providing a simple skeleton for user-written simulations is from CMU's POOMAS package (POOr MAN's Simula).

Two examples of use are given, with different styles of output. One produces a discrete "event log" report, and the other a quasi-continuous "logic analyzer" display.

It is hoped that this paper will encourage programmers and engineers to make more use of simulations in their designs, as they realize that there is no particular magic involved.

An Adaptable Object Code Optimizer for UNIX Systems

Bill Appelbe and Bob Querido

University of California, San Diego
Department of Electrical Engineering and Computer Sciences, C-014
La Jolla, CA 92093

-o-

NCR Corporation, SE-TP
11010 Torreyana Rd.
San Diego, CA 92121

(UUCP: sdcsvax!bill, sdcsvax!querido; ARPA: wappelbe@NOSC)

ABSTRACT

An adaptable object-code optimizer, **xas**, has been developed based upon the UNIX[†] assembler **as**. The optimizer performs both branch optimizations and machine-dependent local optimizations. Local object-code optimizations are defined by means of symbolic templates and actions, similar to the approach used by **yacc** to define grammar productions and actions. The optimizer can be used in conjunction with any compiler which generates assembler, and can readily be extended to perform optimizations which are difficult or impossible to achieve with single-pass code generators such as those used by **cc**, the portable C compiler.

The majority of C compiler implementations are based upon the portable C compiler [Joh78]. Although the portable C compiler can be tuned to generate efficient object code for individual expressions, it often generates poor code for sequences of expressions and flow of control, as code is generated in a single pass. For example, the C program fragment

```
int i,j,k;
...
j = i*i;
k = j+1;
...
```

when compiled by our C compiler generates the following code:

[†]UNIX is a Trademark of Bell Laboratories.


```

...
l4      sp,0(fp)          /* Push i onto the stack */
l4      sp,0(fp)
mr      sp,sp
st4     sp,4(fp)          /* Pop j from the stack */
l4      sp,4(fp)          /* Push j onto the stack */
arl     $1,sp
st4     sp,8(fp)          /* Pop k from the stack */
...

```

In the above example the second 'push' of *i* is redundant, and could be replaced by a duplicate-top-of-stack instruction. Also, the 'push' of *j* is unnecessary, and could be replaced by a duplicate-top-of-stack instruction prior to the 'pop' of *j*.

```

...
l4      sp,0(fp)          /* Push i onto the stack */
dtos
mr      sp,sp
dtos
st4     sp,4(fp)          /* Pop j from the stack */
arl     $1,sp
st4     sp,8(fp)          /* Pop k from the stack */
...

```

By adding more complexity to the C compiler tree-tourer, special cases such as the first optimization could be performed. Since the code for each statement is generated independently, the portable C compiler can not easily perform the second optimization. Of course, a skilled C programmer could have helped the compiler by writing

```
k = (j = i*i) + 1;
```

but other local optimizations, and branch optimizations such as eliminating unnecessary jumps to procedure prologue code¹ require major reorganization of the portable C compiler.

There are three different approaches to improving the code quality of the portable C compiler:

- (1) rewrite the compiler;
- (2) add an additional 'optimization pass' for the intermediate code generated by the first pass of the compiler;
- (3) add an additional 'optimization pass' for the assembler code generated by the last pass of the compiler;

¹ Most C compilers delay generating procedure prologue code until code generation for the procedure is complete. Thus two redundant jumps across the procedure body code are generated in order to execute the possibly void prologue code!

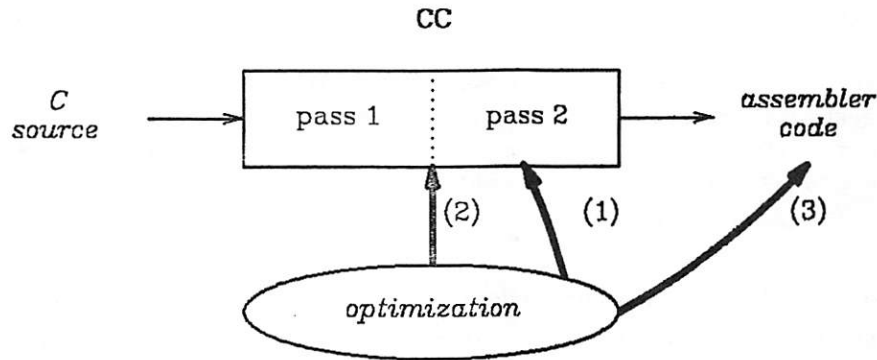


Diagram 1: Incorporating Optimization in the Portable C Compiler

Rewriting the compiler is a major chore, especially since several compilers (**f77**, **pascal**, and **cc**) share the same intermediate code format and second pass. Merely 'adding new templates' to the existing second pass can only provide better local optimization within expressions.

Adding an additional intermediate-code 'optimization pass' can provide global optimizations such as eliminating common subexpressions and moving invariant computations out of loops. However, such an approach is constrained by the organization of the second pass, which generates code for each expression tree in sequence. In addition, developing an intermediate-code optimizer capable of global optimization is a major software project, often requiring person-years of effort [Pow83] [Ank82].

The traditional solution to generating better code has been to incorporate a peephole optimizer as a final pass for the object code. Such peephole optimizers usually consist of a collection of 'hard-wired' machine-specific tests and replacements for inefficient code sequences. The major limitations of traditional peephole optimizers are:

- Difficult to modify or reuse for new UNIX ports,
- Very difficult to incorporate optimizations which require reorganizing blocks of code, or complex analysis of instruction sequences, and
- Difficult to debug or test for new optimizations, or selective levels of optimization.

We have developed an object code optimizer which is considerably more powerful than traditional peephole optimizers, but which is also relatively easily extended to include new optimizations. The goals of the optimizer design and implementation were as follows:

- Low development cost
by making heavy use of existing UNIX tools (powerful optimizers have traditionally been major projects).
- Adaptability.
The design of the optimizer should be machine independent, and should provide a simple interface to enable new optimizations to be incorporated.
- Powerful.
Some optimizations, such as detecting common subexpressions and saving their values in registers, can require repeated passes. Most peephole optimizers are not designed to perform such optimizations. However **xas** can perform arbitrarily many passes, matching patterns against instruction sequences and executing C optimization routines which can include calls to library routines which delete, replace, and insert instructions.

- Systems Programmer Oriented.

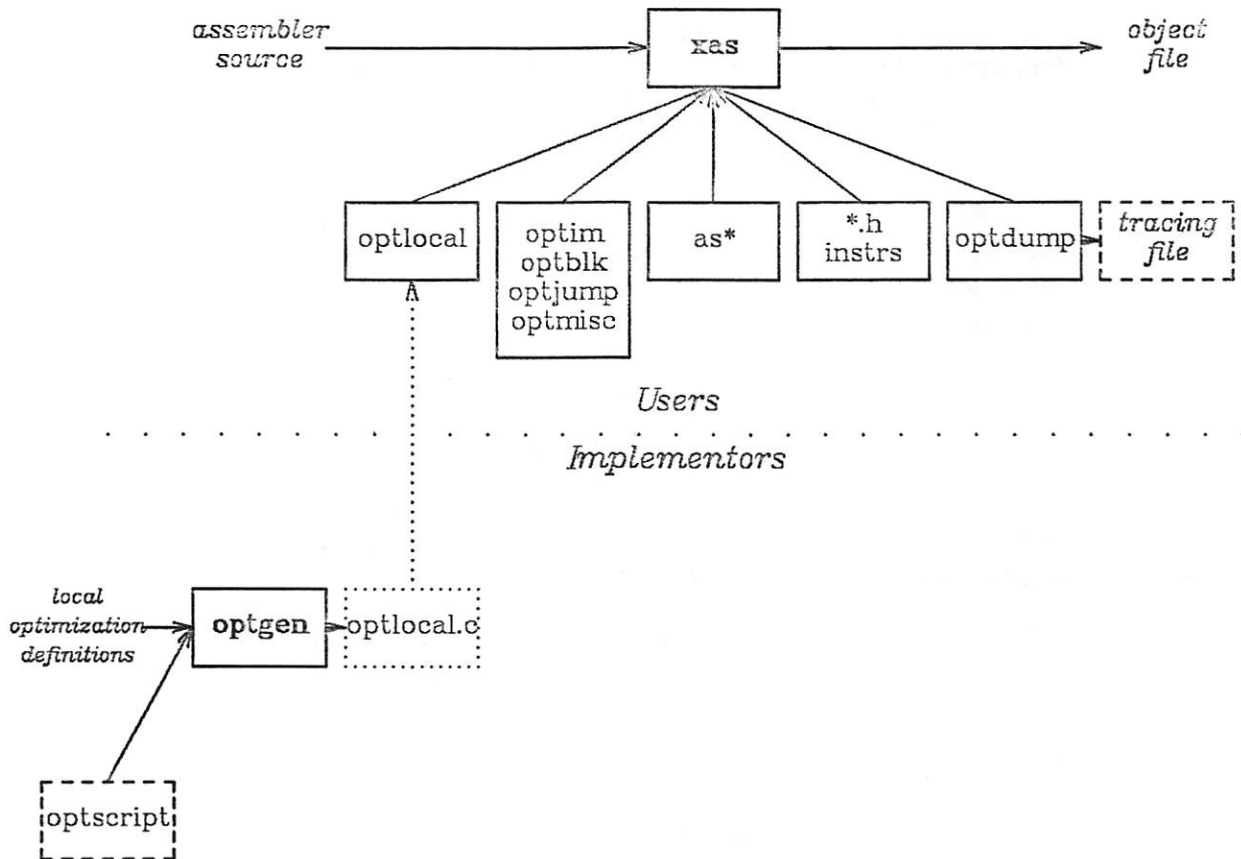
Most systems programs are already optimized to make use of C and UNIX facilities such as register variables, conditional expressions, and system calls for string operations. Our goal was not to attempt to perform such 'user-level' optimizations,¹ as the best way to optimize poorly written code is to rewrite it! Rather **xas** was oriented to performing optimizations inaccessible to the user.

The optimizer is based upon the existing UNIX assembler, **as**, which has been modified to:

- Build a doubly linked list of machine code prior to output of the object code.
- Determine the basic blocks and jump targets, to perform machine-independent branch optimizations.
- Perform multiple passes over the code looking for *instruction patterns*, then performing *optimization actions* whenever a match is successful.
- Finally, the optimized object code is output.

The code to define instruction patterns and optimization actions is produced by a *local optimizer generator*. The generator's design is based upon **yacc**, as the generator takes as input a list of productions consisting of assembler instruction sequences. Associated with each production is an optimization action, written in C. The organization of the optimizer, **xas**, and the local optimizer generator, **optgen**, is illustrated in diagram 2 below:

¹ A tool such as **lint** could be developed to train users to write optimized C (**c_cure?**)

Diagram 2: Organization of **xas**

1. Implementation of Xas

Xas can be regarded as an extension or replacement for the existing System V assembler, **as**¹. The last pass of **as** outputs object code bytes sequentially by means of a call to a function *putins*. **Xas** replaces the call to *putins* by a function which builds a doubly linked list of relocatable object code for each instruction. The next pass of **xas** builds the program flowgraph, then performs machine-independent branch optimizations, including:

- removing orphan blocks of code (unreachable code);
- moving blocks of code which do not immediately follow a **parent** block;
- eliminating jumps-to-jumps, jumps to the following instruction, and 'empty' blocks of code.

After branch optimizations are complete, local optimizations are performed by the routine *optlocal* as follows:

¹ **Xas** and **as** can be maintained together by means of **#defines**.

optlocal

```

optlocal()
{
    ...
    /* Select each local optimization pass */
    for( ; (pass = first_pat->pass) <= last_pass; first_pat = pat ) {
        /* Select each patterns in the pass */
        for( pat = first_pat; pat->pass == pass; pat++ ) {
            /* Select each basic block */
            for( block=0; block<=n_blocks; block++ ) {
                /* Try to match the pattern against
                   each instruction in the block */
                ...
                reset:
                i = Blocks[block].first;
                i_count = Blocks[block].numinsts - pat->length;
                match = 0;
                for( ; i_count-- >= 0; i=i->next ) {
                    if (try_match (pat, i)) {
                        match++;
                        switch (pat->index)
                    }
                }
                if ((match > 0) && (pat->recursive))
                    goto reset;
            }
        }
    }
}

```

/****** Action Code for each pattern goes Here! */

Pattern-Matching Driver for Local Optimizations

Patterns consist of a sequence of object code instructions, in which fields are either constant or *undefined*. The routine *try_match* succeeds if each field of the pattern matches the corresponding field in the instruction sequence, is undefined. If *try_match* succeeds, then the action code corresponding to the optimization pattern will be executed.

This approach allows patterns to be selectively invoked, by grouping patterns into passes which can be optionally invoked when **xas** is used. In addition, sequences patterns can be used to recognize and perform optimizations even when the affected instructions are not adjacent. For example

```

l4      sp,X
...
l4      sp,X
can be replaced by
l4      sp,X
dtos
...

```

provided that none of the intervening instructions in the block are loads or stores. This optimization can be performed by a pattern which first builds a list of loads, followed by a pattern which inserts stores, followed by another pattern which matches loads and uses the data gathered by the previous pattern matches to eliminate unnecessary loads.

2. Optgen — Generating Local Optimizations

The goal of the local optimizer generator was to provide a simple but powerful tool for generating local optimizations. Each local optimization consists of a pattern followed by an action to be performed when the pattern matches a sequence of instructions.

Patterns consist of a sequence of assembler instructions, consisting of an opcode and operands. Fields can be either assembler mnemonics, constants, or undefined (indicated by '?'). Patterns can use the full spectrum of **as** mnemonics, but constant expressions and default operands are not provided. Actions consist of arbitrary C code, which can symbolically access the fields and instructions which matched the pattern. For example, the following optimization replaces redundant loads:

```
store_then_load :
    ST4  ? , DA ? ( ? ) [ ? ]
    L4   ? , DA ? ( ? ) [ ? ]
    = { if( ( $1.5 == $2.5 ) &&
           ( $1.3 == $2.3 ) &&
           ( $1.4 == $2.4 ) ) {
        /* Both instructions reference the same memory address */
        if( $1.1 == SP ) {
            /* Operand is on the stack, so duplicate it */
            insert_before ($1, DTOS, 1);
            if( $2.1 == SP ) {
                /* Result is also on the stack */
                delete_inst ($2);
            }
        }
        else {
            /* Pop the stack */
            replace_inst ($2, LR, ACRR, $2.1, SP);
        }
    }
    else {
        /* Replace the load with a register-to-register load */
        replace_inst ($2, LR, ACRR, $2.1, $1.1);
    }
}
```

Local Optimization

3. Performance

Xas was developed on a VAX-11/780 as a cross-assembler. The optimizer was developed in conjunction with the NCR software development facilities at Torrey Pines, San Diego, and targeted initially for a 32-bit UNIX system under development using the NCR-32 proprietary VLSI chip set. Typically, peephole optimizers achieve mean run time reductions in the range 6 to 50%, depending upon the sophistication of the code-generator preceding the optimizer. Our peephole optimizer yields run time performance improvements of around 20%.

A major disadvantage of optimizers is their overhead at compile time. In our prototype optimizer we have made no attempt to reduce the compile time overhead of using the optimizer. Compilation time overheads could be reduced by sorting patterns into a tree, so that every instruction was not matched against every pattern [Lam 81].

4. Conclusion

The total programming effort to develop the object code optimizer has been relatively small yet the optimizer captures both local and control flow optimizations. Systems programmers have been provided with a tool that enables them to incrementally construct an object code optimizer to bridge the gap between the limitations of the portable C compiler and the perversity of the machine instruction set.

References

- [Ank82] Anklam, P., Cutler, D., Heinen Jr., H., and MacLaren, M.D., *Engineering a Compiler: Vax-11 Code Generation and Optimization*, Digital Press, 1982.
- [Joh78] Johnson, S.C., "A Portable Compiler: Theory and Practice", *Proc. 5th. ACM Symp. on Principles of Programming Languages* (January 1978).
- [Joh78] Johnson, S.C., "A Tour Through the Portable C Compiler", *UNIX Programmer's Manual, Seventh Edition, Volume 2B* (January 1979).
- [Lam81] Lamb, D.A., "Construction of a Peephole Optimizer", *Software - Practice and Experience*, Vol. 11, 639-647 (1981).
- [Pow83] Powers, G., "A Global Optimizing C Compiler", *UNICOM Conference Proceedings*, San Diego, January 1983.

Using Modula-2 for System Programming with Unix

Michael L. Powell

Digital Equipment Corporation
Western Research Laboratory
4410 El Camino Real
Los Altos, CA 94022

ABSTRACT

The C programming language has grown in popularity along with its primary host, the Unix timesharing system. C provides both a well-integrated interface to Unix and a reasonably portable implementation language for the Unix kernel and utilities.

Programming language and compiler advances of the last decade have left C behind. Modula-2 is a language that can incorporate the good points of C while overcoming many of its problems.

The DECWRL Modula-2 implementation fits into the Unix environment well. Modula-2 code may be integrated with C code in the kernel. Modula-2 also allows convenient and substantially safer access to the Unix system calls and libraries.

The flexible type system, intermodule checking, and clean syntax of Modula-2 provide facilities that are increasingly missed in C. The DECWRL Modula-2 compiler also provides code optimization and dependency-based recompilation.

1. Introduction

One of the strengths of Unix [Ritc78] has been how well it is integrated with its primary programming language, C [Kern78]. Unix is an easily modified, extended, and ported system because it is written in C. C is a useful programming language because it has such convenient access to the Unix system facilities.

However, C is a low-level programming language. It provides few of the features that have become common in languages in the last decade, and provides only minimal

protection for programmers who make mistakes. Compared to the previous alternative for systems programming, assembly language, C was a giant step forward in the early 1970's. However, the state of the art of programming languages has moved forward, and C has not kept up. Today, the defenders of C sound like the defenders of assembly language did a decade or more ago.

This paper proposes Modula-2 (as implemented at Digital Equipment Corporation's Western Research Laboratory [Powe84a,Powe84b]) as an alternative to C in the Unix environment. The remainder of this section discusses the good and bad points of C. The next section provides an overview of Modula-2. Following that, we present some extensions to Modula-2 to make it more suitable for system programming in the Unix environment. We conclude with a brief discussion of the compiler and some examples of Modula-2 code.

1.1. The C Domain

In considering a replacement for C, it is important to understand the domain in which the language is used. C is used for three major categories of software: kernel modules, utility programs, and user programs. That C has done well in all three categories is a tribute to its designers. Kernel modules require facilities for circumventing the traditional data type constraints in order to access low-level hardware and operating system structures. Utility programs need convenient access to the operating system interface. User programs must be able to access libraries of useful software and allow programs to be written without intimate knowledge of the implementation or underlying system. C to some extent provides all these things.

Although much of the Unix kernel is like any other software, the hard parts are those that perform privileged instructions and manipulate data whose format is constrained by the hardware. C provides the ability to do address calculations and a good set of operations for extracting and manipulating data. On the other hand, the C kernel programmer must have a good idea what instructions are generated by various constructs. For example, the I/O control status registers may be set as shown on the left, but not as shown on the right.

#define GOBIT 1	struct {
	int gobit:1;
int *csr;	} csr;
*csr = GOBIT;	csr.gobit = 1;

For utility programs, the most important facility of C

is its access to the Unix system calls. A convenient way to parse the command parameters, the ability to access all system services including process creation and interrupts (signals), and a choice of several levels of file operations, combine to make simple tasks easy and even quite complicated programs possible. As far as the C programming language is concerned, the system interface looks just like procedure calls. Unfortunately, since C checks neither the number nor types of procedure parameters, a program that misuses the interface can easily get into trouble.

Although C is one of the better languages for operating system and utility programming, it leaves much to be desired as a general user language. Its control structures are powerful, but its type system omits some common features. Subrange data types are not supported, arrays are always indexed from zero, and array parameters provide no information about their size. Even an operation as simple as a string assignment must be programmed, although there are library subroutines available. Floating point operations, according to the language definition, are always performed in double precision, increasing the expense of numerical computation. Nonetheless, its simplicity and typically small and fast compiler has made C popular for user programming as well.

That C is not perfect in all areas is to be expected. Other popular languages fail in one or more of these areas. Pascal as defined is acceptable only as a user programming language. Fortran has separate compilation but only primitive data structures. Although Ada may eventually be used in all these environments, it is a much larger and more complicated language.

One reason why most languages fail to span all three areas is that their type systems are either too rigid, prohibiting the things that the kernel must sometimes do, or too loose, not providing adequate safety for user programmers. Modula-2 allows the programmer to choose the level of enforcement appropriate to the task at hand.

1.2. Problems of C

The most severe problems of C are in its syntax and its type system. The language syntax permits common errors to go undetected. An if statement such as

```
if (i = 0) {
    printf("i is zero");
}
```

appears to compare *i* with 0, but in fact assigns 0 to *i*. Two-dimensional arrays must be specified as shown on the left.

array[i][j]

array[i,j]

The expression on the right has a valid, but quite different, meaning (it computes the expression *i*, discards it, then uses *j* as the first subscript). Introducing extra semicolons in control structures usually causes no compiler complaint, but can radically change the meaning of the program. The following "loop", for example, executes the statement between the braces only once because of the extra semicolon on the first line.

```
for (i=0;i<N;i++);
{
    printf("i=%d",i);
}
```

It is not only programmers unfamiliar with C who make such mistakes. The problem is that such small differences in the program text produce programs of unlikely correctness that the compiler accepts. Even among correct programs, small syntactic differences in unexpected places can obscure the intended operation of the program.

The syntax for declarations is difficult to parse, either by a compiler or a human reader. The meaning of some declarations, although experimentally determinable using a compiler (not that all compilers always agree), is often the subject of debate. Expressions with side-effects and statements used as expressions often result in hard to read programs. The textual pre-processor that is used for constants and macros often introduces errors as well.

Compared to more modern languages, C is dangerous. It is easy to make small mistakes doing ordinary things, causing problems than manifest themselves in obscure ways. No language will prevent all bugs, but C, like the assembly language it replaced, provides more opportunities for errors than is acceptable today.

2. Modula-2

Pascal is a language that is a contemporary of C with a different approach. It has an elegant type system, but assumes that the entire programming environment is under the control of the compiler and runtime library. In particular, as originally designed, the entire program must be compiled at once. Conceived as a language for students, Pascal omitted the mechanisms for extension that are necessary for system programming.

In many ways, Modula-2 is the confluence of ideas in Pascal and C. It has Pascal's basic data types (with a few

improvements), but also permits types to be changed and addresses to be manipulated. Like C, it has a separate compilation facility, but it provides intermodule typechecking.

2.1. Basic features

A Modula-2 program or implementation module looks much like a Pascal program. Many keywords are common, but some improvements have been made. For example, control structures are fully bracketed so that it is not necessary to add a begin and end when more than one statement is placed in a loop. A loop statement has been added, which allows multiple exit statements. The case statement includes an else clause. Goto statements have been eliminated. Statements may refer to constants, types, variables, or procedures that are later in the compilation, allowing procedures and variables to be defined in any order.

The Pascal data types are present, with a few additions. Unsigned integer data may be manipulated with the data type cardinal. Procedure variables are permitted, although nested procedures may not be assigned to them. A procedure parameter may be declared as an open array, that is, a one-dimensional array of some type with arbitrary size. The procedure can determine at run time the size of the array actual parameter. Thus a procedure can be written to manipulate any size array.

User programmers will see only a few cosmetic changes to the base language. Most of the changes are in the separate compilation facility or in the features for system programmers.

2.2. Modules and separate compilation

With the exception of the main program, programs are organized as modules, each of which has two parts. The definition part of a module (called a definition module) describes those constants, types, and variables, and the interfaces to those procedures that are accessible outside the module. The implementation part of a module (called an implementation module) describes those constants, types, variables, and procedures that are usable only by the module itself, and the code for those procedures that are accessible outside the module.

Each (definition or implementation) module is stored in a separate file. Definition modules are grouped in directories similar to C header files. Implementation modules are like C source files, and are compiled to produce object modules. The resulting object code may be placed in libraries or linked in the normal way.

A module provides a name space for identifiers. By

default, identifiers in one module are distinct from those in another. A module that wishes to allow other modules to use an identifier "exports" that name. Other modules "import" the name from that module. Thus the first few lines of a definition module mentions those things that other modules may use from this module, and the first few lines of a definition or implementation module mentions what things from which modules are used by this module.

When a module is compiled, all of the definition modules that it references must be available so that the types of imported objects may be determined. The implementation modules for those definitions might not even exist. Descriptions of the imported and exported objects are placed in the object modules, and these are checked for consistency before linking.

A module may choose to export a data type without specifying its representation. A so-called opaque type may be used to declare variables in other modules, and those variables may be assigned and passed to procedures, but no other operations may be performed on them. This weak form of data abstraction is adequate for many programming tasks.

Although there is no initialization of variables as in C, modules do have initialization sections that are executed when the program begins. Normally, a module's initialization is executed before the main program starts but after the initialization sections of any modules that it imports.

In the DECWRL Modula-2 compiler, it is not necessary for the implementation of a module to be written in Modula-2. The interface to the Unix system calls and library routines is through a definition module that specifies the types of parameters to the procedures. No interface routines are necessary, since the procedures can be called directly from Modula-2 code, and new procedures may be added easily.

In summary, Modula-2 provides a separate compilation facility with definitions separated from implementations and with the ability to check data types across modules. The module structure organizes the name space, makes explicit which modules depend on what parts of other modules, and provides the means of describing non-Modula-2 software.

2.3. System programming features

Most of the system programming features of Modula-2 are provided through a pseudo-module called system. This module is built into the compiler, and allows certain unusual features to be accessed in a controlled way. Modules that use system programming features must include import statements from the system module.

The contents of the system module is implementation dependent, although the features described below are available in all implementations. The address data type is an untyped pointer that is compatible with all pointers as well as cardinal arithmetic operations. For example, a procedure that has an address formal parameter may be passed any pointer type. There is also a function adr that returns the address of a variable.

The word data type is most useful as a parameter. A formal parameter of type word may accept any one-word data type. A formal parameter of type open array of words may be passed any type variable, and the size of the open array specifies to the routine how large the variable is. The DECWRL Modula-2 compiler also includes a type called byte that is analogous, but provides resolution to the byte.

The size of a variable or type may be determined by calling the size or tsize functions, respectively.

Modula-2 also includes a mechanism for multiprogramming within a single program. The so-called process type is used to represent a coroutine within the program. When a Modula-2 process is created, it starts running on an independent stack. Control is transferred explicitly to another process, using the transfer procedure. It is possible to write a module that implements more elaborate processes with a dispatcher. Full-fledged, independent, memory-sharing processes are conceivable in the language, although they are not yet supported by Unix.

The type of a variable or expression may be changed (even without importing from the system module) by using the type name as a function. The following is an example of such a type change.

```

type
  PtrA : pointer to RecA;
  PtrB : pointer to RecB;
var
  pa : PtrA;
  pb : PtrB;
begin
  ...
  pa := PtrA(pb);

```

Memory may be allocated either through the familiar new and dispose operations, or using routines called allocate and deallocate, which allow the user to specify how much space to allocate. There may be multiple storage allocators, and the new and dispose operations call whichever allocate and deallocate are available at the point of the call.

The above features supply the basis for system programming in Modula-2. Programs can manipulate data independent of its type, and may change the type of data in a controlled way. Programmers, through special modules, their own modules, and modules implemented in other languages, can access the basic machine and language facilities. However, some capabilities, such as control of the representation of storage and convenient interfacing to other languages, are still missing. In the next section, we discuss some of the extensions we have implemented in the DECWRL Modula-2 compiler to address these other problems.

3. Extensions

As defined, Modula-2 omits some features taken for granted in most user and system programming languages. Nonetheless, the structure of the language allows extensions to be integrated as real or builtin modules in a convenient and largely portable way. Occasionally changes have been necessary to the syntax, however. When reserved words were added, the "@" character was used to mark them as non-standard.

The first section describes two modules for I/O and bit manipulations. The second describes system programming and language interface extensions. After that, we describe some other planned enhancements. In all cases, we have tried to maintain what we perceive is the philosophy of Modula-2.

3.1. I/O and Bit Operations

The I/O facility proposed by Wirth is awkward and inadequate for general use. Although interesting in that it uses the open array and word data type to implement a module totally within the type system that does general I/O, it rates low in usability because the compiler provides no information about the types of variables (e.g., a character could be written as a real number) and because the procedures are restricted to a fixed number of parameters.

The io module implemented in the DECWRL Modula-2 compiler is patterned after the printf/scanf facility of C. A format string specifies how the parameters should be interpreted, and an arbitrary number of variables may be specified in one procedure call. The compiler provides type safety by scanning the format string and checking that the types of the parameters match it. The io module provides operations for the typical operations on files, and also permits formatting the information into strings or scanning it from them.

Bit operations such as shift, exclusive or, field extract, etc., are implemented in a module called bitoperations. They are not intended to replace normal arithmetic

or set operations, but there are some applications in which a variable shift or exclusive or can be handy. The bit-operations module is built into the compiler so that these operations may be implemented efficiently (usually a single instruction). However, in other implementations, it could be a normal module.

3.2. Controlling data representation

When in a self-contained environment, it is possible for the compiler to control how all data is stored without the programmer's knowledge. In a multi-language, hardware dependent environment, however, the programmer must instruct the compiler exactly how data is to be stored. This end is accomplished through additional attributes on types, parameters, global variables, and procedures.

By default, an integer, enumeration, or subrange thereof occupies a whole word; a character or boolean occupies a byte. The default size of objects of a particular data type may be overridden by the @size attribute, e.g.,

```
type
  UShort = @size 16 cardinal;
  Short = @size 16 integer;
  SafeUShort = @size 16 [0..65535];
  SafeShort = @size 16 [-32768..32767];
  Bit = @size 1 boolean;
  CSR = record
    go : Bit;
    function : @size 5 @align 1 Function;
  end;
```

Note that SafeUShort will have range-checking applied to it, whereas UShort will not. The @size attribute may prefix any type, including in a record declaration, as in the definition of the function field, above.

By default, variables or fields will be aligned on multiples of their sizes rounded up to a power of two bits. Arrays and records are aligned according to the strictest alignment of their elements or fields. This default may be overridden by specifying the @align attribute. Variables and fields of that type will then be aligned on a multiple of the specified number of bits. Specifying @align 1 will pack fields as closely as possible.

3.3. Interacting with C and Pascal

Since we do not expect to rewrite all software in Modula-2, provisions have been made for Modula-2 code to coexist with C and Pascal code. There are three potential problems: the naming of external symbols, the compatibility

of data passed as parameters, and the sharing of dynamic data structures.

Normally, a Modula-2 procedure or variable that is exported will have a global name of the form `module_name`, where `module` is the name of the exporting module, and `name` is the name of the procedure or variable. (At the level of the loader, as in C, all global names are preceded by an `"_"`.) C and, with a trivial change to the compiler, Pascal programs may refer to such variables directly. Specifying the `@external` attribute, causes the global name to be generated without the module qualifier. For example, the Unix system calls are specified as `@external`, so the compiler generates the correct name when calling them.

The `@size` and `@align` attributes of types described above allow data types to be defined that mimic those in C. Thus it is possible to create corresponding types for each C data type.

There are more subtle issues in parameter passing, however. In C, scalar and structure parameters are passed by pushing the values onto the stack, but array parameters are passed by reference. In Berkeley Pascal, all value parameters are pushed on the stack. In Modula-2, multi-word parameters are passed by reference, with the called routine performing the copy if necessary for value parameters. Although it has not been necessary so far, if needed, a parameter attribute could be added to interface to C or Pascal routines.

Pointer data types are another problem area. Storage allocated by the new procedure is preceded by a check word that allows an efficient and moderately strong pointer validation. A different pointer validation is done by Pascal, and no pointer validation is done by C. By adding attributes to the pointer declaration, a particular type of pointer will be allocated using the Pascal or C allocation routines and will have the proper validation performed when checking is enabled.

3.4. Further extensions

Two additional features are planned for the near future, both relating to arrays. The open array parameters are currently restricted to passing the whole array, and only one dimension of the array type may be open. The natural extension of this is to permit multi-dimensional open array parameters and to permit the actual parameter to specify a cross-section of the array. This represents only a small change to the syntax and the implementation of open arrays.

The second feature to add is support for dynamic

arrays, arrays whose bounds are determined at runtime. These are "open array variables", and are treated like pointers to arrays. Normally, the array bounds will be available so runtime checking is possible.

4. The DECWRL Modula-2 Compiler

The DECWRL Modula-2 compiler has been described elsewhere [Powe84a,Powe84b]. The compiler is easily retargeted since it generates a dialect of P-code that has run on the Cray-1, VAX, Z8000, and M68000. It contains a machine independent optimizer that eliminates common subexpressions, moves invariants out of loops, and allocates registers based on weighted static reference counts. The code generated by the compiler for the VAX is comparable to (and often better than) the best compilers for the VAX. It substantially outperforms the C compilers available on Unix.

The intermodule checker is also interesting, in that it will automatically recompile modules that are out of date. This recompilation is based, not on timestamps of the source files, but on the actual type dependencies. If, for example, a new procedure is added to a definition module, only the modules that use that procedure will be compiled.

The compiler is relatively small, approximately 23,000 lines of Pascal. The compile speed is not as fast as the Unix C compiler, but the speed will improve when the compiler is translated into Modula-2.

5. Summary

The C programming language has shown that progress in programming languages is not PL/1 (nor Ada). Modula-2 appears to have benefitted from the C philosophy, but also adds a measure of safety not available in C. Modula-2 is new to the Unix environment, and some features are evolving that will make it better suited to it. The DECWRL Modula-2 compiler is an attempt to make Modula-2 be a worthy successor for C.

6. References

Kern78

B.W. Kernighan and D.M. Ritchie, "The C Programming Language," Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

Powe84a

M.L. Powell, "Modula-2: Good News and Bad News," Proc. Spring 1984 IEEE COMPCON.

Powe84b

M.L. Powell, "A Portable Optimizing Compiler for Modula-2," Proc. SIGPLAN Compiler Construction Conference, June 1984.

Ritc78

D.M. Ritchie and K. Thompson, "The UNIX Timesharing System," Bell System Technical Journal 57, 6 (July/Aug. 1978), pp. 1905-1930.

Wirt82

N. Wirth, "Programming in Modula-2," Springer-Verlag, New York, 1982.

7. Examples

7.1. Bit map module

The following module creates a bit map data structure. Notice the @nocheck attribute that disables pointer validation, the use of the system and io modules, the open array of words parameter, and the two kinds of storage allocation, new and ALLOCATE.

file bitmap.def

```
definition module bitmap;
from system import Word;
export qualified BitMap, MakeBitMap;

type
  UShort = @size 16 Cardinal;
  BitPointer; (* opaque type *)
  BitMap = pointer to BitMapRec;
  BitMapRec = record
    bitmap : BitPointer;
    width, height : UShort;
  end;

procedure MakeBitMap(width, height : Cardinal;
  bits : array of Word) : BitMap;
  (* bits may be anything *)
end bitmap.
```

file bitmap.mod

```
implementation module bitmap;
from io import terminal, Writef;
from system import TSize, MAXINT;
from Storage import ALLOCATE;

type
  BitPointer = pointer @nocheck to set of [0..MAXINT];
  WordPointer = pointer @nocheck to
```

```

        array [0..MAXINT] of Integer;

procedure MakeBitMap(width, height: cardinal;
    bits : array of word) : BitMap;
var
    bp : BitPointer;
    wp : WordPointer;
    bm : BitMap;
    size, i, wordsize : Cardinal;
begin
    New(bm);
    wordsize := (width * height + TSize(Word) - 1)
                div TSize(Word);
    ALLOCATE(bp, wordsize * TSize(Word));
    bm^.height := height;
    bm^.width := width;
    bm^.bitmap := bp;
    wp := WordPointer(bp);
    for i := 0 to wordsize-1 do
        wp^[i] := bits[i];
    end;
    Writef(terminal, "MakeBitMap:");
    for i := 0 to wordsize-1 do
        Writef(terminal, " %08x", wp^[i]);
    end;
    Writef(terminal, "0");

    return bm;
end MakeBitMap;

end bitmap.

```

7.2. Unix interface

The following is an excerpt from the Unix interface module and an example of its use. Notice the open array parameters without counts, the use of @external, the access to the Unix system calls and the io module.

file unix.def

```

definition module unix;
from system import Byte;
export qualified
    errno, open, close, read, fork;
type
    CString = array @nocount of Char;    (* open arrays *)
    CBuffer = array @nocount of Byte;    (* without counts *)
var
    @external errno : Integer;    (* shared global variable *)

procedure @external open(name : CString; mode : Cardinal)

```



```

        : Integer;
procedure @external close(fildes : Integer);
procedure @external read(fildes : Integer; buffer : CBuffer;
        nbytes : Cardinal):Integer;
procedure @external fork() : Integer;
end unix.

```

file testunix.mod

```

module testunix;
import io;
from unix import open, close, read, fork;

procedure Connect() : Integer;
var
    ttyName, ptyName : array [1..100] of Char;
    childpid, pty, stderr, i : Integer;
    c : Char;
begin
    loop
        io.SWritef(ptyName, "/dev/pty%c%x", c, i);
        pty := open(ptyName, 2);
        if pty >= 0 then
            io.SWritef(ttyName, "/dev/tty%c%x", c, i);
            stderr := open(ttyName, 2);
            if stderr >= 0 then
                exit;
            else
                close(pty);
            end;
        end;
        inc(i);
        if i > 15 then
            i := 0;
            inc(c);
            if c > 'z' then
                return -1;
            end;
        end;
    end;
    childpid := fork();
    (* ... *)
    return childpid;
end Connect;
end testunix.

```

The FP-Shell

Manton Matthews and Yogeesh Kamath

Department of Computer Science

University of South Carolina

Introduction

One of the strongest features of UNIX is that it provides a framework in which the development of tools is encouraged and in which these tools can be easily combined to make other tools. A common file structure and the pipe construct allow several programs to be connected in a linear fashion with each performing a portion of the overall task and passing its results to the next program. This framework has encouraged the development of a large number of well designed tools, which in turn has encouraged programming at the command language level[5,6,16]. In many cases the shell programming languages have been used for rapid prototyping of systems. These prototypes typically combine many off-the-shelf Unix tools to perform the desired task. This ability to reuse software by combining existing programs into new programs is one of the primary goals of the Functional Programming (FP) languages introduced by Backus[1,2]. FP languages use program forming operations (PFOs) to combine existing programs to form new ones. The Bourne shell "pipe" construct acts (in the absence of side effects) as the PFO composition. The FP-Shell extends the function combining ability of the Bourne Shell to include the other PFOs (apply-to-all, construction, etc.) of the example FP language given by Backus[1]. Also, the FP-shell provides us with an FP interpreter that we have used for research on FP algorithms and for instructional purposes. Our interpreter compares favorably in several ways to the one developed at the University of California at Berkeley [4].

FP Systems

Functional Programming systems were first introduced by Backus[1,2]. One of the primary goals of these systems is to lower software development costs by facilitating the reuse of existing software and by easing the burden of software maintenance. In these systems programs are true mathematical functions and there are functional operators which can be used to combine existing programs into new programs. The mathematical nature of these systems makes the verification of software a reasonable goal. This, along with modularity in design, will make modifications to existing systems easier, more reliable and less frequent.

A Functional Programming (FP) System consists of:

- (1) A set of objects
- (2) A set of functions that map objects into objects
- (3) A single operation, application of a function to an object, denoted by $f : x$
- (4) A set of functional forms or program forming operations PFOs which are used to construct new functions from existing functions

(5) A set of definitions of functions and the mechanism for defining new functions.

The atoms are a subset of the set of objects that typically includes the integers, character strings, reals, the booleans T and F, etc. An object then, is either an atom, a finite sequence $\langle x_1, x_2, \dots, x_n \rangle$ of other objects, or the special object bottom, denoted by \perp . In the original presentation of FP systems by Backus, bottom was used to represent any error condition, including non-terminating computations. In our system \perp represents only non-termination; all other errors are represented by error message objects [14].

Examples of Objects

1.732 $\langle 1\ 2\ 3 \rangle$ $\langle \langle 1\ 2\ 3 \rangle\ \langle 4\ 5\ 6 \rangle\ \langle 7\ 8\ 9 \rangle \rangle$ $\langle a\ b\ \langle abc \rangle \rangle$

The set F of functions include a subset of primitive functions. Other functions can be built up using the PFOs and definitions. All functions are bottom preserving ($f : \perp = \perp$ for all functions f). The set of primitives will usually include arithmetic functions, data manipulators for selecting portions of and rearranging objects, and a collection of predicates.

Examples of Functions and Applications

addition	$+$: $\langle 12, 34 \rangle = 46$
transpose	trans : $\langle \langle 1\ 2\ 3 \rangle\ \langle 4\ 5\ 6 \rangle \rangle = \langle \langle 1\ 4 \rangle\ \langle 2\ 5 \rangle\ \langle 3\ 6 \rangle \rangle$
tail	tail : $\langle a\ b\ c\ d \rangle = \langle b\ c\ d \rangle$
selectors	3 : $\langle a\ b\ c\ d \rangle = 3$ 2 o 1 : $\langle \langle a\ b\ c \rangle\ \langle d\ e\ f \rangle \rangle = b$
relational operators	le : $\langle 3\ 4 \rangle = T$ eq : $\langle 3\ a \rangle = F$
concatenation	concat : $\langle \langle 1\ 2 \rangle\ \langle 3\ 4 \rangle\ \langle 5\ 6 \rangle \rangle = \langle 1\ 2\ 3\ 4\ 5\ 6 \rangle$
append-left	apndl : $\langle x\ \langle a\ b\ c\ d \rangle \rangle = \langle x\ a\ b\ c\ d \rangle$

For a list of the other primitive functions in the FP-shell the reader is referred to [1,8].

The set of PFOs determine the power of an FP system. The sample FP system that Backus presented and the FP shell include the following PFOs:

(1) composition $f \circ g : x \equiv f : (g : x)$

(2) construction $[f_1, f_2, \dots, f_n] : x \equiv \langle f_1 : x, f_2 : x, \dots, f_n : x \rangle$

(3) apply to all $\alpha f : \langle x_1, x_2, \dots, x_n \rangle \equiv \langle f : x_1, f : x_2, \dots, f : x_n \rangle$

(4) conditional

$$(p \rightarrow f ; g) : x \equiv \begin{cases} f : x & \text{if } p : x = \text{TRUE} \\ g : x & \text{if } p : x = \text{FALSE} \\ \perp & \text{otherwise} \end{cases}$$

(5) insert

$$/f : x \equiv \begin{cases} x_1 & \text{if } x = \langle x_1 \rangle \\ f : \langle x_1, /f : \langle x_2, \dots, x_n \rangle \rangle & \text{if } x = \langle x_1, x_2, \dots, x_n \rangle \text{ and } n \geq 2 \\ \perp & \text{otherwise} \end{cases}$$

(6) while

$$(\text{while } p \text{ } f) : x \equiv \begin{cases} x & \text{if } p : x = \text{FALSE} \\ (\text{while } p \text{ } f) : (f : x) & \text{if } p : x = \text{TRUE} \\ \perp & \text{otherwise} \end{cases}$$

- (7) constant $\%x : y \equiv \begin{cases} \perp & \text{if } y = \perp \\ x & \text{otherwise} \end{cases}$, where x is an object parameter to this functional form.
- (8) binary-to-unary $(b u f x) : y \equiv f : \langle x, y \rangle$, where x is an object parameter to this functional form.

The choice of the set of primitive functions and functional forms determines the power of an FP system. For weak sets of primitives and functional forms it might not be possible to compute all computable functions.

Examples of FP Programs

(1) Inner Product of two vectors

Def innerprod = ($/ +$) $\circ (\alpha \times)$ \circ trans .

The definition of this FP version of inner product [1] corresponds more closely to the way we think about inner products than does say a program in Fortran, Pascal or C. The transpose function pairs up corresponding elements of the vectors, then the $\alpha \times$ multiplies each of these pairs, and finally insert + sums the results.

(2) Quicksort

```
def quick = (small -> id;
             concat  $\circ$  [quick  $\circ$  lelist, [1], quick  $\circ$  gtlist]  $\circ$  [1, tail, %<>]) .

def gtlist = (null  $\circ$  2 -> 3; (gt  $\circ$  [1  $\circ$  2, 1] ->
                             gtlist  $\circ$  [1, tail  $\circ$  2, apndl  $\circ$  [1  $\circ$  2, 3]]);
                             gtlist  $\circ$  [1, tail  $\circ$  2, 3])) .

def lelist = (null  $\circ$  2 -> 3; (le  $\circ$  [1  $\circ$  2, 1] ->
                             lelist  $\circ$  [1, tail  $\circ$  2, apndl  $\circ$  [1  $\circ$  2, 3]]);
                             lelist  $\circ$  [1, tail  $\circ$  2, 3])) .

def small = le  $\circ$  [length, %1] .
```

In the application of this quicksort program to a list, if the list has less than 2 elements then the predicate small will evaluate to TRUE and the value of the quick function will be obtained by applying the identity function (id) to the argument. However, if the length is greater than 1, then quick splits the list up into three lists containing respectively the elements less than or equal to the first element, the first element, and the elements greater than the first element. Quick is called recursively on the nontrivial sublists and finally all three lists are concatenated to give the sorted list. The function gtlist takes as argument an object with 3 components the first is a number, the second is a list of numbers left to compare, and the third component is a list of numbers that are greater than the first component.

Functional View of the Unix Shells

The Unix shells, the Bourne Shell, C-shell, KSH and others[5,7,10], provide the framework necessary for productive combination of existing programs into new commands. In fact, this framework is provided by the underlying operating system and these shells only realize the potential provided by Unix. The components of Unix that have contributed to this framework correspond to components of functional programming systems. This is not to say that the shells are FP systems, because the "functions" of the shells can have side effects, i.e. they are not true mathematical functions. But, there are substantial similarities between the two systems.

The most significant correspondence between the Unix shells and FP systems is that the pipe construct corresponds to the functional form composition. For example consider the following example:

```
tbl paper | eqn | vtroff
```

and consider the corresponding command written in a more FP-like syntax:

```
vtroff o eqn o tbl : paper
```

The only differences are in the order of specifying the functions and the symbol used to denote the composition (also the ":" that is used in the FP version.)

In FP systems the programs are true functions and this is one area, as we have pointed out earlier, that the standard shells fail to be like an FP system. However, to be able to use the combining capability provided by the shell, the user is encouraged to write filters, programs with a single input and a single output. Filters map files into files in the way that FP functions map objects into objects. Pipes are possible between two filters if the first program produces an output file which has a format that is acceptable to the second program as input. In the same manner the composition of two FP functions is defined if the first function produces an object to which the second function can be applied. Thus, the Unix shells have an FP subsystem consisting of filters and the pipe combining operator. In this subsystem files correspond to objects, with the atoms being bytes.

Another area of similarity between Unix and FP systems is in the design of certain tools there is a restricted form of the apply-to-all functional form built into the tool. For instance the cat command operates on any number of specified files in the way $\alpha \text{ cat} : < \text{file}_1 \text{ file}_2 \dots \text{file}_n >$ would work.

Design and Implementation of the FP-Shell

Our primary objectives in developing the FP-Shell were:

- (1) to provide a framework which is easy to understand and modify for the study of functional systems,
- (2) to extend the ability of the Unix shells to combine programs by including other functional forms,
- (3) to investigate applications in which the extra combining capability is utilized and gather usage statistics.

User commands are broadly divided into two categories, FP expressions and system commands. FP expressions are evaluated by converting the input command string into a binary tree representation and manipulating the representation in an appropriate way. The FP-Shell uses a top-down recursive descent LL(1) parser since it is easy to implement and has good error detection properties. Semantic evaluation is accomplished by traversing the tree and executing procedures corresponding to the operations indicated in the current node. After the completion of each application, the tree is reconfigured into a new form releasing the function and object nodes not required for further evaluation.

In the study of alternative FP languages it is important to be able to add new primitives, to add new functional forms, and to modify the syntax of existing forms. In the FP-Shell the addition of new primitive functions is quite easy, since it requires no modification to the parser. Also, since Lex is used to generate the lexical analyzer, the only modification required to the front end is the addition of a pattern in the Lex specification file. The addition of new functional forms may require modifications to the underlying grammar and this type of additions is not as simple. We have added two new functional forms to the FP-Shell, tree insert [19] and parallel-apply.

tree insert

$$(tree\ f):x \equiv \begin{cases} x_1 & \text{if } x = \langle x_1 \rangle \\ f : \langle (tree\ f) : \langle x_1, \dots, x_m \rangle, (tree\ f) : \langle x_{m+1}, \dots, x_n \rangle \rangle, & \\ & \text{if } x = \langle x_1, x_2, \dots, x_n \rangle \text{ and } m = \left\lfloor \frac{n}{2} \right\rfloor \\ \perp & \text{otherwise} \end{cases}$$

parallel-apply

$$PA(f_1, \dots, f_m):x \equiv \begin{cases} \langle f_1 : x_1, f_2 : x_2, \dots, f_m : x_m \rangle & \text{if } x = \langle x_1, x_2, \dots, x_m \rangle \\ \perp & \text{otherwise} \end{cases}$$

The addition of tree insert was easier because it is syntactically like regular insert and thus did not require major modifications to the parser. Another type of alteration to the FP-Shell that we plan to consider is variations on the FP syntax. This investigation will include consideration of variations on the syntax of FP to see if they improve program clarity. For example, should compositions be written in the usual mathematical order (last function to be applied listed first) or the reverse order (which is used by the Bourne Shell)?

In the evaluation of an FP expression when the interpreter encounters a function name it is first assumed to be an FP primitive that is built into the system. If this is not the case then the function is assumed to be one of the user defined functions that has been loaded into the workspace. If the function is neither a primitive nor a user defined function then the system uses the PATH variable to search for a file, a Unix command or FP-Shell script, with this name. When an FP-Shell script is encountered it is automatically loaded into the workspace and then interpreted. If the same name is encountered again it will already be in workspace. The FP-Shell considers Unix commands, such as cat, tbl, sort, Csh and Bourne shell scripts, etc., as primitive functions of the FP system. These primitives are interpreted by making system calls to execute the particular process. The creation of new processes (child processes) is accomplished with the 'fork' system call.

One of the goals of the FP system is to achieve parallel computation with functional forms such as 'construct' and 'apply-to-all'. Interpretation of such applications is done by creating individual child processes for each inner applications and executing them in parallel. At the end, the main process checks the results for any errors. Individual buffers for input and output for these processes are created by the 'pipe' system call.

Another important aspect of the integration of the FP-Shell within Unix is that files are interpreted naturally as FP objects. The information in the files which users can retrieve and store, is maintained in a simple sequential character stream. With many of the most useful Unix filters (such as grep, sort, sed and awk) files are logically subdivided into lines. In the FP-Shell this logical subdivision of files is carried further, with a files being interpreted as FP objects comprised of sequences of lines, with lines interpreted as sequences of words and words as sequences of characters. A file 'TEST' with n lines is interpreted as:

TEST = $\langle \langle \text{line}_1 \rangle, \langle \text{line}_2 \rangle, \dots, \langle \text{line}_n \rangle \rangle$
 with $\text{line}_1 = \langle \langle \text{word}_1 \rangle, \langle \text{word}_2 \rangle, \dots, \langle \text{word}_p \rangle \rangle$
 and $\text{word}_1 = \langle c_1, c_2, \dots, c_q \rangle$

Thus, the computation $2 \circ 1 : \text{TEST}$ would be defined and the result would be the second word of the first line.

In the presentation of FP-systems, Backus represents all errors generated as a result of the application undefined functions, undefined operations and nonterminating operations by \perp . Since this is inadequate for debugging the FP-Shell represents all errors except nontermination by error objects, which are appropriate error messages. In addition to the regular objects the FP-Shell includes a special set of objects, the error objects. For each primitive function there is a collection of error atoms, which are error messages describing the various errors in the application of the primitive. Error objects then are sequences of objects that contain one error object as a component. Indication of errors are not limited to primitive function applications. In a large application, errors from the preceding applications (if any) are noted and are indicated along with the error message of the current application. When an error object is returned as the value of a computation, it is automatically saved and an error message is printed. The level of the error message is selectable by the user and can vary from a message containing just the error atom of the primitive that caused the error, to a full trace of the sequence of function calls that led to the error.

The FP-Shell as an Interpreter

The FP-Shell provides us with an interpreter that we have used for research on computer architectures and algorithms for FP and for instructional purposes. The other FP interpreter available to us was developed at the University of California at Berkeley, by Scott Baden [4]. Our FP interpreter includes several features that are in the Berkeley implementation, including a trace facility, and a mechanism for gathering execution statistics. Both interpreters are based on the FP language presented by Backus.

In the Berkeley FP interpreter, all errors in computations are represented by \perp , which is denoted by '?'. With this approach, Baden and Patel state that debugging of programs is made very difficult [4] and propose a trace facility to help in debugging. We have extended the FP system to include useful error messages as described earlier. In addition to providing useful error messages, the interpreter offers several other improvements over the Berkeley interpreter.

- (1) The FP-Shell is implemented in C and requires only 45K of text space and 18K data space. The Berkeley interpreter is written in Franz Lisp and requires 121K text space and 645K data space. Thus, the FP-Shell requires less space, is slightly more portable and is available on machines with 64K address space, provided they have separate instruction and data space capability.
- (2) The FP-shell is slightly faster than the Berkeley FP interpreter.
- (3) The degree of integration into Unix is greater with the FP-Shell because any Unix command can be used as a primitive within any FP expression.
- (4) In the Berkeley implementation the mechanism for gathering execution statistics counts the number of times each function is executed and in some cases provides the length of the argument to which it is applied. The FP-shell provides a more general mechanism for estimating the execution time of an algorithm. The user can supply a subroutine for each primitive function that will accurately estimate the execution time for that primitive function on a proposed architecture. The default routines provided by the system gather statistics similar to those of the Berkeley interpreter.

In addition the FP-Shell offers the following features:

- (1) The FP-Shell provides system commands for defining data objects and a mechanism for saving the results of any computation in an object, which can be used in later computations.
- (2) The FP-Shell provides the ability to edit functions or objects in the work space. Edited functions or objects are automatically reloaded.

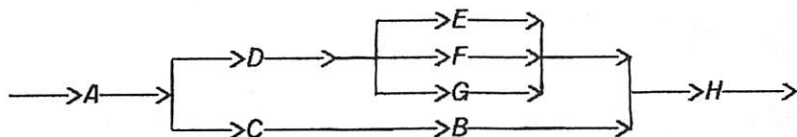
- (3) Standard libraries may be included by simply inserting the key word '#include' with the name of the library in the program file. An option of making copy of a loaded library or just the name of the library is provided while saving the workspace.
- (4) Another useful system command is the session copy command. This option when set provides a hard copy of the session from the time it is given. This feature has been particularly useful for students and instructors to evaluate their performance.

Area of applications

In general we can say that the introduction of programming constructs like "apply-to-all" and "construct" enhances the capabilities of a command language. The usefulness of the apply-to-all functional form as a general mechanism should be apparent to the users of Unix utilities (such as cat and grep) that operate on any number of files specified in the command line. This capability would be provided by the functional form and would not have to be built into each of the tools.

The extended combining capability of the FP-Shell provides a richer environment for very high level language programming applications. For instance an image processing environment can be developed in which the images are represented as FP objects. The FP-Shell provides a useful tool manipulating images on UNIX files. Powerful algorithms to process information in the matrix form may be written easily by combining already powerful primitives such as transpose, distribute and rotate.

In a multi-processing environment or in a multi-language computer network, the combining capabilities of an FP-Shell can be utilized to specify data communication paths [15]. For example consider the data transformation routed through a network of processors as shown below:



The output of this transformation may be specified in FP as:

$$H \circ [[E, F, G] \circ D, B \circ C] \circ A : \text{input}$$

Conclusions

The system was originally implemented on a PDP-11/44 in C and has now been ported to a VAX-11/780 and a 68000 based microcomputer in the Department of Computer Science, at the University of South Carolina. Currently the FP-Shell provides us with an excellent environment for studying FP systems. Before it can be considered an excellent shell a few of the standard shell features need to be included. Among these are file name expansion, command and variable substitution, a history mechanism (including command editing and retry facilities) and job control.

REFERENCES

- [1] J. Backus, "Can Programming Be Liberated From the Von Neumann Style? A Functional Style and its Algebra of Programs," CACM, Vol. 21, No. 8, August 1978, pp. 613-641.

- [2] J. Backus, "Function-Level Computing", IEEE Spectrum, Aug. 1982, pp. 22-27.
- [3] S. Baden and D. Patel, "Berkeley FP - Experiences with a Functional Programming Language", COMPCON Spring 1983.
- [4] S. Baden, FP User's Manual, University of California, Berkeley.
- [5] S. R. Bourne, "The UNIX Shell", Bell System Technical Journal, Vol. 57, No. 6, part 2, July-August 1978, pp. 1971-1990.
- [6] T. A. Dolatta and J. R. Mashey, "Using a Command Language as the Primary Programming Tool," in Command Language Directions: Proc 79 IFIP Working Conference on Command Languages, D. Beech, ed., North-Holland, Amsterdam, The Netherlands, 1980.
- [7] W. N. Joy, An Introduction to the C Shell, Department of Electrical Engineering and Computer Science, University of California, Berkeley.
- [8] Yogeesh H. Kamath, "FP-Shell, A Functional Programming Environment for UNIX", M.S. Thesis, University of South Carolina.
- [9] B. W. Kernighan and R. Pike, The Unix Programming Environment, Prentice Hall, 1984.
- [10] D. Korn, "KSH - A Shell Programming Language," Proc. Summer 1983 USENIX Conf., Toronto, pp. 191-202.
- [11] J. Levine, "Why a Lisp-Based Command Language?", SIGPLAN Notices, Vol. 15, No. 5, May 1980, pp. 49-53.
- [12] J. A. Malcom, Brevity and clarity in Command Languages, SIGPLAN Notices, Vol. 16, No.10, Oct. 81, pp. 49-53.
- [13] M. M. Matthews, "Functional Aspects of Unix", IEEE SOUTHEASTCON '83 PROCEEDINGS, pp. 91-94.
- [14] M. M. Matthews, "Introducing Good Error Message Capabilities into an FP System" TR 83-005 Department of Computer Science, University of North Carolina at Chapel Hill, Aug. 1983.
- [15] J. H. Morris, "Real Programming in Functional Languages," in Functional Programming and its Applications, ed. J. Darlington, P. Henderson and D.A. Turner, Cambridge Press 1982, Cambridge, pp. 129-176.
- [16] D. M. Ritchie, "The UNIX Time-Sharing System: A Retrospective", Bell System Technical Journal, Vol. 57, No. 6, part 2, July-August 1978, pp. 1947-1970.
- [17] J. Shultis, "A Functional Shell", SIGPLAN Notices, Vol. 18, No.6, 1983., Proc. Symp. on Prog. Lang. Issues in Software Sys., pp. 202-211.
- [18] L. A. Stabile, "FP and its Use as a Command Language", Proc. COMPCON, 1980.
- [19] J. H. Williams, "Notes on the FP Style of Functional Programming," in Functional Programming and its Applications, ed. J. Darlington, P. Henderson and D.A. Turner, Cambridge Press 1982, Cambridge, pp. 73-102.

SYSTANT*:
An Integrated Programming Environment
for Modular C
under UNIX**

Stowe Boyd
Research/Development

AZREX, INC.
3 Mountain Rd.
Burlington, Ma. 01803-4799
[Usenet address: wivax!cadmus!azrex!gizmo]

1. Introduction

SYSTANT is a UNIX-based integrated programming environment (IPE) designed to support medium to large programming systems. The environment is geared strongly toward the phases of the software life cycle following specification analysis and initial modular design.

SYSTANT is essentially a single language system, designed for Modular C [Boyd83] although a straight-forward means of including C language [Kern78] libraries is incorporated into both the environment and target language. A minimal subset of the system, intended solely for training purposes, is available for C users.

The SYSTANT environment is tightly allied to methods of top-down structured design, modular decomposition of systems [Parn72] and the use of abstract data types [Lisk74, Boyd84a], although no rigid methodological restrictions are placed upon the system's use. SYSTANT achieves a uniform and consistent model of system dynamics by adhering to these principles. Major system components include a dedicated, customizable shell (s7); a metaprogramming toolset (s4); documentation and analysis aids (sect.s 8 and 9); on-line help facilities (s9); design and management aids (s6); and a extensible collection of reusable modules (s3).

2. Modular C

The C language is widely used in systems programming due to its flexibility and machine-level orientation. C fails to provide a high degree of type checking [Appe84] and this weakness makes programming-in-the-large an almost unreachable goal [Ichb79].

*SYSTANT is a trademark of AZREX, Inc.

**UNIX is a registered trademark of Bell Laboratories.

2.1. Language Description

Modular C is an extension to C which incorporates an explicit syntactic parallel to C's intrinsic modularity. Although Modular C is more strongly typed than C, it can be compiled by standard C compilers. Translation of Modular C into C is performed by the C preprocessor and does not require pre-compilation, unlike some C variants [Cox83, Katz83, Stro82]. A Modular C PARSER/DEBUGGER System currently under design will enforce strong typing restrictions fully.

Modular C has the features necessary for the implementation of abstract data types [Bovd84a]: independent compilation of "multi-procedural modules" [Lisk74], and restrictive access to module components.

A Modular C program is a collection of independently compiled modules. Modules may not be nested, and access to module components is highly constrained. Module specification is uncoupled from implementation, as it is in Ada [Ichb79], and the operations made visible by a module must be referred to via attribute notation*.

Module components fall into three groups: data abstraction constructs (or abstractors), inherited data information (or inheritors), and function bodies (or bodies).

Abstractors define attributes of the module: operations, exceptional conditions, variables and type information. There are three kinds of Abstractors: interface, visible and hidden. A module has at most one visible and one hidden abstractor; the visible one is translated into an interface. A module's visible attributes are externally referenced through its interface abstractor, and internally referenced through the visible. A module's hidden attributes may not be referenced externally.

A module may import interfaces from many other modules in order to exploit their capabilities. All function bodies are inaccessible directly. A run-time prelude binds function bodies to their respective definitions within an abstractor, and this prelude may be used to initialize variables as well. In sum, modules implement a controlled means of managing resources.

Inheritors distribute types and defined constants across certain Modular C domains. PUBLIC inheritors provide hierarchical sharing across multiple, perhaps un-interfaced, modules. PUBLIC inheritors are extra-modular, and must be managed by the programming environment. PACKET inheritors are products of specific modules, yielding data information to those modules requesting interface privileges**. PRIVATE inheritors are strictly local, and their information is

* stack.push(x), for example.

** Using type coercions, the equivalent of Ada's private and limited private types are implementable.

available only within the local module context. SYSTEM inheritors (s3) are Modular C descriptions of run-time libraries. Like PUBLIC inheritors they are extra-modular and maintained by the programming environment, designed for checking library calls and their parameters.

A function body is either an OUTBODY, bound to the module's visible abstractor, or an INBODY, bound to the hidden abstractor. An OUTBODY is an entry point to the module's capabilities, and bodies collectively implement the module's functionality.

2.2. Exceptions

A structured mechanism for defining, raising, trapping and guarding against exceptional conditions has been implemented, in a manner not dissimilar to that of CLU [Lisk79]. An optional trap may be placed at the close of a body's scope, in which handlers for exceptions may be defined. Modules define the exceptions which may occur during its operations; each operation defined has an optional clause for this purpose. In calling a module's operations, there must be explicit guards for exceptions, and a trap entry should be designated to handle each exception. (Verifying traps for semantic completeness will be a responsibility of the PARSER/DEBUGGER System.)

2.3. Control and Expressions

The control statements of C have been modified to disambiguate the curly brace overloading: keywords like LOOP, ENDLOOP, SCOPE, ENDSCOPE, IF and ENDIF have been introduced for this purpose. AND and OR are used for boolean operators, and all language keywords must be upper case. Other C control and expression features have been retained, so that at the expression level Modular C is nearly identical to C.

2.4. Modular is Better

If modular design is both a productive tool and a powerful means of dealing with large programs, then, by the same token, a language geared toward modularity will be productive and powerful. There is a consensus that such languages are needed, but those available are often too esoteric, unreliable or monolithic to be effective. Modular C relies on the portability, flexibility and power of the C language, erecting a modular front-end to the C core. The two, when coupled, offer a better solution.

3. Reusability

Generating reusable modules causes a designer's time-frame to lengthen and perspectives to shift. Long-term group benefits will take precedence over short-term costs in time and effort, since a well designed module may be effectively reused by many later projects. In effect, the environment's value to the user increases with respect to the volume of available, practical, reusable modules.

In order to stimulate reusable design the programming environment must reduce the costs of reusability, and provide simple means of inspecting, testing and binding reusable modules. SYSTANT supports very high levels of code reuse, and minimizes the overhead involved in its production.

3.1. Catalogs

Within SYSTANT catalogs of modules may be created, accessed, extended and browsed. New developments may bind previously implemented modules and abstract layers (composed of collected modules) in a well managed way. The binding of a compiled image is transparent to the user, although its interface is linked to the developing (requesting) module. This permits full intermodular type-checking, similar to that of the INTERCOL [Tich79] and Cedar [Teit83] systems.

3.2. Systems

C libraries may be extended, providing full type checking, by the creation of system specifications. All of the standard UNIX C libraries have been defined in the current prototype of SYSTANT. These definitions do not require modification of existing code instances. Systems are one of Modular C inheritor constructs (s2.)

4. Metaprogramming, Views and Relations

A high proportion of contemporary IPEs are centered around syntax-directed editors [Habe78, Teit81, Reis84], which have explicit knowledge of the language syntax, and whose commands are primarily tree-oriented. In principle, such editor systems speed the development process by catching syntax and type errors upon entry, and providing the user with multiple views of the program being developed. In fact these systems are liable to be inflexible (some eliminate convenient manipulation of the program as text), slow, and impractical for large programs due to time/space constraints. While such efforts will, most likely, lead to practical syntax-oriented environments, currently they do not.

An alternative IPE design used in SYSTANT, and other IPEs [Baye80, Chea80], is based upon the manipulation of program components by system provided operations. This has been called "metaprogramming" [Chea80, Boyd84b] due to its similarity to data manipulation within programming languages.

SYSTANT provides a well integrated set of manipulators for SYSTANT components. These components include all of Modular C's constructs, as well as documents, specifications, and other programming artifacts. A create/update/forget* paradigm is used throughout SYSTANT. Relations of the various Modular C constructs remain consistent due to propagation of attributes throughout a network of managed text objects.

* SYSTANT will not destroy objects. At most, it is willing to "for-

A sequence of manipulators, interpretable by a UNIX or SYSTANT shell program, can implement the complete structure of a module, or any SYSTANT managed object. Here "complete structure" denotes both abstract and logical aspects of the object. Parallel to the logical representation are any number of abstract representations, in the form of documentation or formal specifications (s9.)

The physical storage layout of these objects on the UNIX file system may be totally transparent to the user, although direct access is available. The SYSTANT SHOW facility provides logical "views" of the components, using an extensible knowledge base; the UNIX utilities, such as "more," "ls" or "cat," show only the physical layout.

5. Architecture, Security and Archives

The basis for the SYSTANT architecture is the UNIX file system: a hierarchical tree structure. SYSTANT maintains tree structures of known depth and complexity, and manages interlinkage of components by means of a distributed group of files: the SYSTANT architectural database. A great deal of the architecture is fully determined, centered around the addition and deletion of named objects of known kind. This rigidity mirrors the inherent structure of Modular C modules, the hierarchical array of modules within a program, or abstract layer, and the directed graph of intermodular coupling. A collection of modules within the system is known as an assemblage, a term which generalizes programs and abstract layers. The terminal objects of the architecture are the concrete representations of Modular C modules, including artifacts like documents; these objects are subordinate to named modules, which are in turn subordinate to named assemblages. Assemblages may implement programs, but may also be used to implement abstract layers in a complex, abstract machine. In either case, assemblages are subordinate to named projects.

This architecture is meant to reflect the human state of affairs: the way we think about programming systems, and the distribution of authority [Daly80].

Assemblages are the site where the compiled images of modules are bound together. Assemblages also are the staging area for the creation of novel modules. They are generally associated with various documents and specifications.

Named projects play a somewhat different role relative to assemblages, since assemblages are not generally created by way of detailed specifications. These top-most nodes are allied with other, more global information: project-wide type definitions and defined constants, and the archive of the project's modules and linkage information.

Although the access mechanisms of the UNIX file system may be

get." UNIX commands may be used to destroy physical objects. SYSTANT can be "reminded," as well.

sufficient for some purposes, they are often not flexible enough to enact usable "power-based" privilege mechanisms [Mins84]. SYSTANT has a higher-level security system, the CHECK system, which provides arbitrary constraints upon particular users or groups relative to logical objects. Explicit "checking-out" and "checking-in" of modules or assemblages is required, once local "checks" have been defined. The system ensures that the meta-activities of the system builders within the context of a "checked" module or assemblage do not transgress the restrictions laid out by its designer/owner. These restrictions are kept within a CHECK database, associating users and groups with specific privileges and constraints. They may be modified by SYSTANT manipulators.

Allied to security and access is the archiving of programming systems. Each SYSTANT project node has an ARCHIVE database associated with it. SYSTANT provides a front-end to source code control systems which manages the SYSTANT hierarchy in a fashion similar to that described by Cristofor [Cris80]; tables which relate particular versions of archived modules are maintained by the ARCHIVE front-end. These tables in turn have versions which are associated within another, higher level table; this hierarchy parallels the architecture. In this way, the project, assemblage and module versions are maintained in a minimal space.

The current ARCHIVE system uses SCCS [Roth75] as a back-end, although almost any code control package could be used.

6. Goal-Directed Programming

Programming is a disjoint process; many aspects of programming blend and overlap in unpredictable patterns. Large programs are not conceived in one moment, leaping full-blown into the designer's mind. The activity of programming is best considered as a series of goals, which are (presumably) reducible to sequences of actions taken by the programmer.

6.1. Goals

Goals in the SYSTANT IPE are tied to the creation and manipulation of logical objects. The CHECK security system updates the GOAL database when triggered by users seeking legitimate access privileges relative to modules or assemblages. The user checking-out a module, for example, may include a goal description as part of the command. The CHECK system enters time, module name, user name, and the goal of the check-out into the database. Perusal of the current goals and goal history of SYSTANT users is available to all users of the system.

Since programmers have higher and lower-order goals, not all activities are logged in the database; it is not a substitute for UNIX accounting. Users may enter goals which are not bound to the CHECK system; such goals are manipulated by their declarers, and are not managed by the CHECK system.

6.2. SCHEMES

The design prototype of a module may be a semi-formal, natural language description of its functionality and constraints [Hest81], or a formal algebraic or axiomatic notation [Hoar69, Lisk75, Beic84]. To support such higher-level design activities, SYSTANT provides the SCHEME system, which is not biased toward any particular methodology.

Subordinate to a SCHEME are PROTOCOLs, which specify visible operations of modules. SCHEMES and PROTOCOLs are managed as text, and manipulators are provided for both.

6.3. Scenarios

Once a designer is convinced of the validity of some logical definition, a scenario of manipulators may be devised to implement the logical framework of the SCHEME. This scenario is in fact a script of shell commands. The designer runs the scenario, and may then inspect the produced SYSTANT object, whether module, group of modules or a collection of module components. If the produced object is defective in some way, it may be destroyed and the design process reinitiated.

7. A Knowing Environment

Throughout SYSTANT, knowledge of the architecture and the implicit relations between objects is used heavily. These knowledge bases are relatively fixed, and are not always readily extendable. One tool* in which the knowledge base is flexible is the SYSTANT shell.

7.1. The Tailorable SYSTANT Shell: SYSTER

The SYSTANT shell, SYSTER, provides the functionality of the standard shells (I/O redirection, pipes, scripts and aliases,) but is remarkably different in several ways. SYSTER has a large SYSTANT knowledge base, and assists the user in simple ways; it makes use of the user's level of expertise, and varies its level of feedback with regard to that level. SYSTER command evaluation follows an on-demand, functional programming style; this makes recursive macros possible.

These features aside, the SYSTANT shell allows users to customize their shell by actually writing modules (in Modular C) which can be loaded into the shell**. In effect, the user can modify (to some extent) the actions of the shell, extend knowledge bases, create novel knowledge bases (unrelated to SYSTANT, for example), implement

* Another flexible knowledge base is found in the SHOW utility, which is charged with providing views of logical objects.

** the mechanism for this will be fully addressed in a forthcoming paper; G. Kotikian and S. Boyd, "SYSTER: The SYSTANT Shell." The method is also used in the SHOW tool.

SYSTER commands, and directly manipulate the tokens managed by the shell. This gives the user the ability to efficiently configure the programming environment to a great degree.

8. Analysis and Status

A large set of analysis tools are bundled into SYSTANT: static analysis of modules, confirmation of intermodular connections, lint-picking and parsing of program fragments, and exception analysis. The analyses are treated as attributes of Modular C objects, and are manipulable by the same paradigms as other attributes. A configuration management tool, CONMAN, performs checks on the architecture and database maintained by SYSTANT.

In addition to these tools, information yielded by many of the SYSTANT tools, such as the GOAL, CHECK and ARCHIVE systems, contribute to fuller analysis.

SYSTANT possesses a STATUS system which maintains status information at each node of the SYSTANT architecture. A large proportion of the SYSTANT tools update a STATUS entry, leaving behind a trace of their work, and a result. Comparison of these STATUS objects and the GOAL data for an assemblage or module can yield information difficult, if not impossible, to derive by other means.

9. Documents, Help and Tutor

Programming documents are important not only for the users of completed programming systems, but also figure prominently in the design process. Association of specialized documents with various aspects of a programming task has been used as a primary design medium [Hest81]. SYSTANT provides documents at the project, assemblage and module nodes automatically, copying user defined forms into those texts.

All SYSTANT principles and user information reside in an INDEX document system, which is managed by SYSTANT. The tools used to construct, maintain and browse the system are available for other applications. The system is hierarchical, and indexed by keywords. Selection can be categorical, or keyword oriented.

A particular instance of the INDEX system is used for on-line help. Users may ask for help by subject, command name, Modular C construct, and receive information and cross-references.

A SYSTANT TUTOR has been designed which will prompt novice users for parameters to match SYSTANT commands. The TUTOR is directly known to the SYSTANT shell, and the novice user may think of it as a SYSTER feature, although it is an independent program.

10. Byline, Timeline and Summary

SYSTANT is a powerful, adaptive and complex system for the production of large programming systems in Modular C. A working subset of

the system is now in use at Azrex, and a larger, more advanced version will be ready for beta release in August, 1984.

We are actively soliciting friendly, local sites as beta environments. The commercial release is scheduled for January 1985, and will be premiered at UniForum at that time.

SYSTANT was bootstrapped out of previous versions of itself, and is entirely written in Modular C.

11. Acknowledgements

I would like to thank Garen Kotikian for his unending efforts, his many contributions and improvements. I also thank Don Gilmore for early valuable insights; Dave McIlhenney and Azrex for supporting the work; Stuart Comer, Scott Williams, Chris Stacey and Jerry Doland for wrestling the juggernaut. To all, many thanks. Finally, thanks to Sarah Hoover for editing this and other, related papers.

12. References

- KEY: CACM: Communications of the ACM;
 SPN: SPN; IETSE: IEEE Transactions on Software Engineering;
 SPE: Software -- Practice and Experience.
 SEE: , H. Hunke, ed. Springer Verlag, 1980.
- [Appe84] W.F. Appelbe and A.P. Ravn, "Encapsulation Constructs in Systems Programming Languages." ACM TOPLS 6, 2, April 1984.
- [Daly80] E.B. Daly, "Organisational Philosophies used in Software Development." Infotech State of the Art Review, 1980.
- [Baye80] M. Bayer, et. al., "Software Development in the CDL2 Laboratory." SEE 1980.
- [Beic84] F.W. Beichter, O. Herzog and H. Petzsch, "SLAN-4 -- A Software Specification and Design Language." IETSE 10, 2, March 1984.
- [Boyd83] S. Boyd, "Modular C." SPN 18, 4, May 1983.
- [Boyd84a] S. Boyd, "Free and Bound Generics: Two Techniques for Abstract Data Types in Modular C." SPN 19, 3, March 1984.
- [Boyd84b] S. Boyd and Sarah Hoover, "Integrated Programming Environments: Metaprogramming." Submitted April 1984, Unix Review.
- [Chea80] T.E. Cheatham, "An Overview of the Harvard Program Development System." SEE 1980.
- [Cox83] B.J. Cox, "The Object Oriented Pre-Compiler." SPN 17, 1, January 1984.
- [Cris80] E. Cristofor, T.A. Wendt and B.C. Wonsiewicz, "Source Control + Tools = Stable Systems." Proc. Compsac 80, October 1980.
- [Feld79] S.I. Feldman, "Make -- A Program for Maintaining Computer Programs." SPE 9, 1979.

- [Habe78] A.N. Habermann, "The GANDALF Research Project." CMU Computer Science Research Review, 1978-1979, 1979.
- [Hest81] S.D. Hester, D.L. Parnas and D.F. Utter, "Using Documentation as a Software Design Medium." Bell Tech. J. 60, 8, October 1981
- [Hoar69] C.A.R. Hoare, "An axiomatic basis for computer programming." CACM 12, October 1969.
- [Ichb79] J. Ichbiah, et al., "Rationale for the Design of Ada." SPN 14, 6, June 1979.
- [Katz83] J. Katznelson, "Introduction to Enhanced C." "Higher Level Programming and Data Abstractions--A Case Study Using Enhanced C." SPE 13, August 1983.
- [Lisk74] B. Liskov and S. Zilles, "Programming with Abstract Data Types." SPN 9, 4, April 1974.
- [Lisk75] B. Liskov and S. Zilles, "Specification Techniques for Data Abstractions." IETSE 1, 1, March 1975.
- [Lisk77] B. Liskov, A. Snyder, R. Atkinson and C. Schaffert, "Abstraction Mechanisms in CLU." CACM 20, 8, August 1977.
- [Lisk79] B. Liskov and A. Snyder, "Exception Handling in CLU." IETSE 5, 6, November 1979.
- [Mins84] N. Minsky and A. Borqida, "The Darwin Software-Evolution Project." SPN 19, 5, May 1984.
- [Parn72] D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules." CACM 15, 12, December 1972.
- [Reis84] S.P. Reiss, "Graphical Program Development with PECAN Program Development Systems." SPN 19, 5, May 1984.
- [Roth75] M.J. Rothkind, "The Source Code Control System." IETSE 1, 4, December 1975.
- [Stro82] B. Stroustrup, "Classes: An Abstract Data Type Facility for the C Language." SPN 17, 1, January 1982.
- [Teit81] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: a syntax-directed programming environment." CACM 24, 9, September 1981.
- [Teit83] W. Teitelman, "Cedar: an interactive programming environment for a Compiler-Oriented Language." LANL/LLNL Conference on Work Stations in Support of Large Scale Computing, March 1983.
- [Tich79] W.F. Tichy, "Software Development Control Based on Module Interconnection." Proc. 4th Int. Conf. on Software Eng., 1979. pp. 29-41.

LIPs : Knowledge Base Development System

Kiyoki Ohkubo

Software Section, Minicomputer Department,
PANAFACOM Limited

2-49, Fukaminishi 4-Chome
Yamato-Shi, Kanagawa-Ken,
242 Japan

ABSTRACT

In this paper I describe an experimental knowledge base development system LIPs, which aims to be the kernel of an integrated environment for a future office information system. LIPs is named after its main components LISP, INGRES^{*)}, and PROLOG, and 's' is for plural form of LIP. But, LIPs isn't a mere complex of these components. Most advanced point of LIPs is a harmonious integration of software technologies (functional programming, logic programming, object oriented programming, relational database, programming environment). Below, I explain new features about each component, and as a whole system. This has been developed in about two years on UNIX^{**)} and written in C and itself(Lisp & Prolog).

1. Introduction

There are two types of knowledge base development system (Hayes-Roth, 1983). The first type is derived from the actual implementation of knowledge base system and extracted the core part which is independent of its application. Such examples are EMYCIN which derived from MYCIN, KAS from PROSPECTOR, and HEARSAY-III and AGE from HEARSAY-II.

The second type is the approach from a language which facilitates knowledge representation capability and inference mechanism. OPS5, EXPERT, ROSIE, RLL, KRL, and FRL are categorized into this type.

LIPs is included in the second type. LIPs provides Lisp for representing procedural knowledge, Prolog for declarative knowledge, C for time critical or highly algorithmic calculation. And huge structured data are able to be stored in relational database.

*) INGRES is a trademark of Relational Technology Inc.

**) UNIX is a trademark of AT&T Bell Laboratories

2. Architecture of LIPs

Fig. 1 shows a conceptual architecture of LIPs.

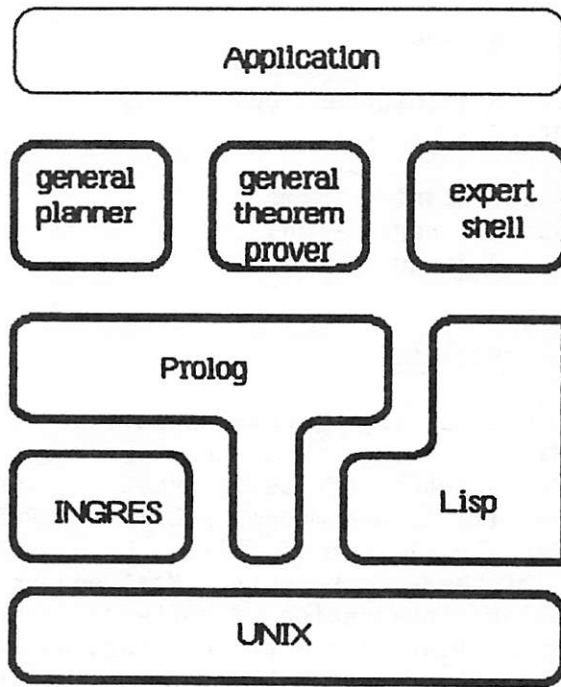


Figure 1: The conceptual architecture of LIPs

Main features of LIPs are:

- (1) Prolog as main language
 - similar syntax with DEC-10 prolog.
 - higher order predicates are available.
- (2) Lisp as deterministic programming language
 - based on LispKit(Henderson, 1980) lisp and has a smart functional feature.
 - Lisp functions are automatically compiled and can be called as Prolog predicates.
- (3) INGRES can be called as Prolog predicates
- (4) Parallel worlds of LIP can be generated.

- parallel execution and/or object oriented programming are easy.

(5) Comfortable programming environment taking merits of UNIX

- interactive terminal monitor which provides editing and calling shell facilities.

The level which comprises of "general planner", "general theorem prover", and "expert shell" are now in researching.

3. LIPs Language Features

3.1. The predicates defined by Lisp

To make a unified environment, we decided LIPs to have Prolog interface as top level, and Lisp can be used for defining body part of prolog statement. An example program is:

```
oblist(X):=
(letrec oblist
(oblist lambda 'nil
  (obliss (get 'oblist 'apval) 'nil '3))
(obliss lambda (x y d)
  (if (zerop d) 'nil
    (if (null x) 'nil
      (if (eq (car x) 'pname) (cons (car (cadr x)) y)
        (let
          (if (null z) (obliss (cdr x) y d)
            (nconc z (obliss (cdr x) y d)))
          (z obliss (car x) y (sub1 d))))))))).
```

This program constructs a list of all atom names, and can be used like any other statements defined in Prolog. Notice the point that the head is Prolog, the body is Lisp, and the neck is intermixed.

One more usage is to speed up a certain case of argument pattern.

```
append(X,Y,Z):=(lambda (X Y) (append X Y)).
append([],[],Z).
append([A|X],Y,[A|Z]):-append(X,Y,Z).
```

The first statement is called only when X and Y are nonvar, and will shorten execution time.

While reading into memory, these statements which have Lisp bodies are automatically compiled and registered as Lisp functions, and then converted into normal Prolog statements as follows.

```
append(X,Y,Z):-lisp(append(X,Y,Z)).
```

The predicate "lisp" checks argument pattern. To connect Prolog and Lisp, there is the second predicate named "prop" which is used for getting or putting a property list.

3.2. The predicates for INGRES

There are predefined predicates which are written in EQUOL (INGRES preprocessor which converts QUEL like statements embedded in a program written in C into function calls). They are used for interconnecting between Lisp & Prolog world and INGRES. The next list shows supported functions.

- (1) invoking INGRES, and consulting the member relations of the "Database".

```
ingres(Database,Relation_name_list).
```

- (2) consulting the attributes of the "Relation".

```
relation(Relation,Attributes_of_relation).
```

- (3) retrieving and asserting tuples as facts.

```
retrieve(Relation).
```

- (4) appending facts to the "Relation".

```
append(Relation).
```

- (5) creating a "Relation".

```
create(Relation,Attributes_of_relation).
```

- (6) destroying a "Relation".

```
destroy(Relation).
```

- (7) exiting from INGRES.

```
quit_ingres(Database).
```

3.3. Creating multi worlds

The first predicate for multi worlds is:

```
para(W1,W2,...).
```

This is like 'or', but fork processes and begin execution parallel in the same environment as the parent process. This results in tree like parent and children relationships as shown in Fig. 2. Each world executes its purpose as cooperating actor. This function has two purposes.

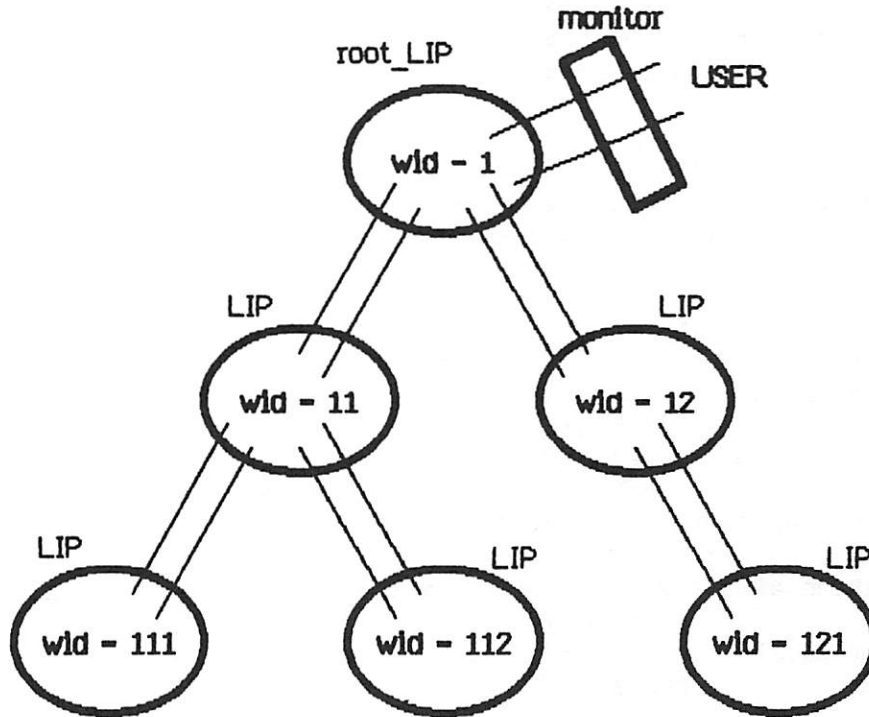


Figure 2: Parallel world of LIPs

- (1) By executing alternative sub-goals simultaneously, we can prevent a situation to fall into an infinite recursive call and die for short of memory.
- (2) As will show in the next sub-section, we can construct communicating parallel world cooperating together to form a blackboard mechanism like HEARSAY-II(Erman, L. 1980). This function also helps the user to develop system with high modularity.

3.4. Unification as communication protocol

Parallel worlds created are assigned world identifier (wid) and communicate with their parent by UNIX pipe. When the world communicates with his sister, first the request is caught by his parent and then send to her.

Communications are done by Prolog unification, a kind of compromise. Not only a read/write level protocol, there is the second predicate for multi worlds:

```
request(Wid,Goal).
```

The receiver specified by Wid unifies the Goal with self environment (instantiate), and send back value to the caller. As such, communication is done by give and take manner.

4. LIPs Programming Environment

Where human cost is high, such as in the development of the knowledge base system, there should be a good environment comparable to it.

The root LIP, which the user invokes as the UNIX command, has a monitoring facilities to help interaction. This function was implemented by getting a hint from the interactive terminal monitor of INGRES. The interaction with the user is usually logged in the editor buffer. There are commands which facilitate editing, redoing (consult or reconsult), reading from a file, writing to a file, calling shell, etc. The editing function is supplied by the internal call of vi (full screen editor developed at UCB), and when a syntax error is detected, the next invocation of the editor will show the screen where it exists. These commands may be entered as a reply to the LIP prompt, and differentiated from usual inputs to Prolog, facts and rules, by the head of the input string. Fig. 3 presents the data flow in root LIP.

Another important component is the debugging tool. When developing applications with Prolog, it is difficult to point out where and how the binding of variables were taken and when it was instantiated, even the program was coded by yourself. This phenomenon is caused by the declarative nature of Prolog. This shows an importance of the debugging environment. In the LIPs system, adding to the usual trace function of a certain range and/or specified predicates, there is a break package which facilitates a single step execution and an explanation against why-type and how-type queries. Why-type queries (why this sub-goal was called?) are answered by showing goal tree toward the root, and how-type queries (how this sub-goal was satisfied?) are answered by showing partially completed sub-tree.

The conditions to be able to make an integrated programming environment easily are the clarity of the language syntax and semantics, and its descriptive power. I propose one criteria of measuring the cost requires. It is the number of program steps of interpreter when written in self language. Simple version of Prolog requires several steps, Lisp requires several ten steps, and C or Pascal will probably requires several thousand steps. Although Prolog is said weak in environment compared to Lisp, it is promising with this indicator.

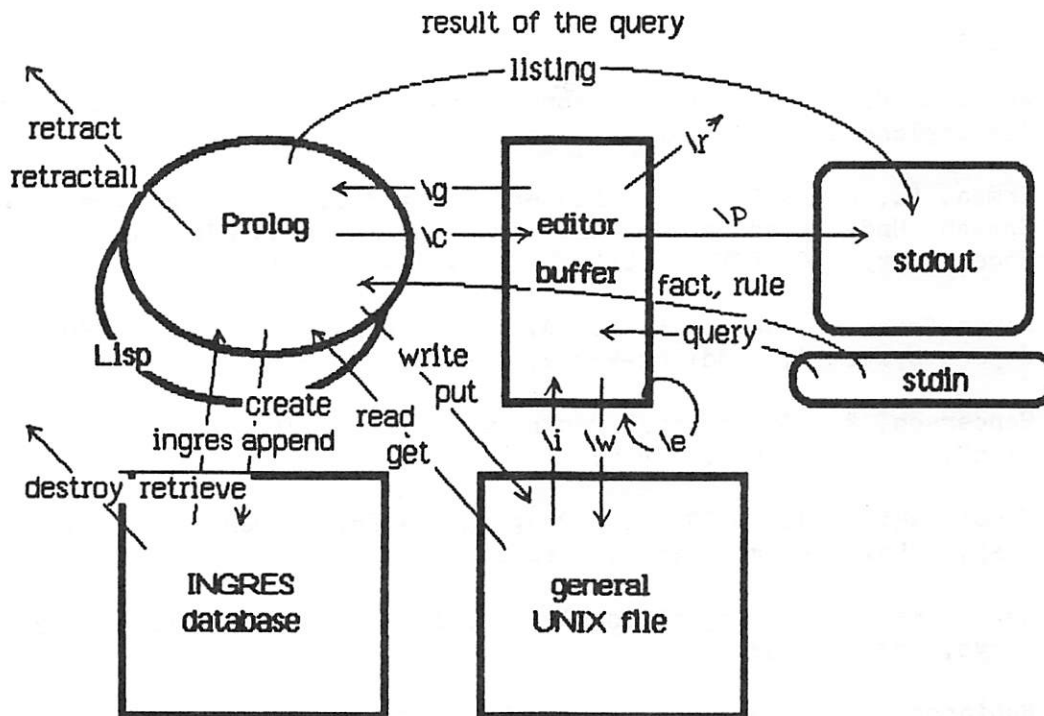


Figure 3: Data flow in root LIP.
The strings beginning with "\" are monitor commands.

5. Conclusion

Similar systems with LIPs are developing around the world. Hybrid feature of lisp and prolog, LOGLISP(Robinson, 1982), QLOG(Komorowski, 1982), Prolog/KR(Nakashima, 1982). Concurrent feature of prolog, Concurrent Prolog(Shapiro, 1983). Object/Actor model, Smalltalk(Adele, 1983), INTERMISSION(Kahn, 1982). These approaches are concerned with finding new foundations which a computer should be ground on. LIPs aims to be another "soft" machine.

Motivation to make LIPs is getting an environment where advanced software against office can be easily developed. For example, natural language interface, planning, scheduling, project management, decision support, prediction, and data interpretation. Expert systems are now

considered to help experts or users who require expert. I think, in near future, expert systems approaches will be applied to office information system. In office, small number of rules in a job, but large number in total job. LIPs will provide a new environment.

References

- 1) Adele Goldberg and David Robson: "Smalltalk-80: The Language and Its Implementation", Addison-Wesley (1983).
- 2) Erman, L., Hayes-Roth, F., Lesser, V., and D.Reddy: The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty, COMPUTING SURVEYS 12(2)213- 253 (1980)
- 3) Hayes-Roth, F., Waterman, D. A., and Lenat, D. B.: "Building Expert Systems", Addison-Wesley (1983).
- 4) Henderson, P: "Functional Programming: Application and Implementation", Prentice-Hall (1980).
- 5) Komorowski, H.J: QLOG - The Programming Environment for Prolog in Lisp, "Logic Programming", Academic-Press (1982).
- 6) Nakashima H.: Prolog/KR User's Manual, METR 82-4 (University of Tokyo, Tokyo, 1982).
- 7) Robinson, J.A. and Silbert, E.: LOGLISP: Motivation, Design and Implementation, "Logic Programming", Academic-Press (1982).
- 8) Shapiro, E.Y.: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report, TR-003 (1983).

WINDX - WINDows for the UNIX Environment

Peter E. Collins

Ithaca Intersystems, Inc.
1650 Hanshaw Road
Ithaca, New York 14850

1. Introduction.

Window systems are generally accepted as providing both a powerful and a natural way for humans to interface to multitasking systems. The difficulty is determining how to implement this facility for a particular operating system, in this case UNIX in all its versions (6, 7, bsd4.2, system III, system V). The system programmer must carefully consider where in UNIX the window system will be created. Should the system be implemented at the higher level by cooperating programs and special shells, or at the device level as a specialized tty driver? In addition the designer must consider how the window software is distributed between the main host cpu and the terminal. This is a critical decision if we are to allow many users to enjoy the window environment without seriously impacting overall system performance and response. The user would also like to be able to run all the usual UNIX utilities. This means we would like to implement windows in a manner that is transparent to applications programs.

These design considerations can be summarized as the following goals for a window system implementation:

- * Efficiency - require little host overhead
- * Transportable - work on any version of UNIX
- * Transparency - allow all programs to run
without special modification

2. Two Previous Solutions

Present systems have tried both high level and low level implementations of windows. An example of the high level approach is BRUWIN[1] which uses shells and cooperating tasks to provide windows on text terminals. The ATT 5620 or Blit terminal on the other hand, represents an

attempt at a lower lever solution.

2.1. BRUWIN.

The BRUWIN system implements a window environment for general text terminals as described in termcap. It is implemented as a collection of programs communicating via pipes. A display manager maintains windows on each terminal, keeping track of window priority and overlap. A virtual terminal emulator task is used to provide programs with a uniform terminal interface by making all terminals behave as if they were a VT-52. Finally, a tasking controller arbitrates application program access to window input and output.

This approach accommodates a large number of text terminals and is fairly portable among UNIX versions. It is reported running under both version 7 and Berkeley UNIX.

BRUWIN is implemented entirely on the host, with no help from the terminal. Unfortunately this places a tremendous burden on the host. When the display manager manipulates a window it may be forced to redraw all windows which involves a fair amount of IO. Not only must the display manager resolve window overlap, it must maintain complete copies of each window's contents. There is 100% duplication of screen information on the host.

2.2. Blit - a low level, distributed approach.

Bit mapped displays are now most commonly used for window system terminals. The Blit terminal[2] is representative of this class of terminal. The window environment for Blit, referred to as layers, is implemented at a lower level in UNIX, at the level of the device driver. It also distributes the windowing support software by downloading what is essentially the display manager and terminal emulator into the terminal. The host need only concern itself with multiplexing application program and terminal IO once the layer environment is established.

Unfortunately, graphics operations, such as line drawing, become complicated in the Blit layer environment. Partially covered layers force special considerations for graphics algorithms. Luckily, these are all handled by the downloaded software.

While distributing the window management task relieves the host of a good deal of work, the layers environment is not transparent. Such significant programs as the C shell will not run. Also, so far, older versions of UNIX have not been supported by the Blit.

2.3. GRAPHOS and WINDX.

Like Blit, GRAPHOS[3] is designed to relieve the host of window management tasks. However, this goal is achieved by a more sophisticated terminal architecture than a simple bit map. By combining the concept of the character cell of the standard text terminal and the bit map of the standard graphics terminal, GRAPHOS presents a hybrid architecture that elegantly supports windows and efficiently utilizes refresh memory to allow maximum offscreen image storage.

With this scheme, the virtual terminal or layer is implemented by a structure called the page. The page is a rectangular collection of 8 by 16 pixel cells that represents a terminal with m rows and n columns, where the product, mn , is less than 32,768. All text and graphics operations occur on some page and can be carried out without concern over what portions of the page may be visible. In fact, these operations can take place on an invisible page.

Pages maintain a complete terminal environment including such terminal attributes as their own color table, their own GKS graphic environment including line styles, transformations matrices, etc, and terminal characteristics including scrolling mode, line wraparound and the like.

The page contents are viewed through screen windows, rectangular regions of the screen that map rectangular regions of the page onto the screen. Many windows may appear on the screen, with the terminal managing the overlapping areas on the screen window level. This allows page operations to completely ignore current window visibility conditions. The screen windows are true windows through which we are able to view portions of virtual terminals inside GRAPHOS.

Since windows are collections of pointers to cells in refresh memory, only one description of each character need be stored in the terminal. This is important because with 4 bits required to store the color information at each pixel it takes 64 bytes to specify a cell's contents. Once the initial refresh memory is set aside to describe the character set, GRAPHOS needs only 2 bytes to reference any character on the page. When graphics operations are performed, cells are allocated as needed from the refresh memory for those portions of the page containing graphics. Since graphics typically contain large blank areas or areas of solid colors this allows GRAPHOS to use the minimum amount of refresh memory for each page. This enables us to have graphics on any page we desire without the necessity for huge amounts of memory to store complete bit maps for each of the 32 possible pages.

A windowing system for GRAPHOS can be implemented at either the high or the low level of UNIX. In either case, the system will distribute the window management task by relying on the terminal's builtin window capabilities. Our first attempt at implementing a UNIX window interface, WINDX, took the high level approach, using cooperating programs and shells modeled after the BRUWIN system. This would ensure that the system could be transported to any UNIX version and would be reasonably transparent to application programs. By distributing most of windowing task to the terminal we believe WINDX does not present a very large burden to the host. Terminal emulation of the VT100 and Tek 4010 terminals was also handled by the terminal so each virtual terminal had a choice of terminal type.

The only task left to implement on the host is directing input from the terminal to the appropriate task and placing task output on the correct page. This is simply a large exercise in IO traffic control. The usual pipe mechanism is used to communicate between the window output manager, the window input manager, and the application tasks. Using pipes also allows us to perform non-blocking reads and writes to the terminal by checking pipe status before each IO operation.

While currently implemented on a Xenix system, WINDX uses only standard features of UNIX to ensure portability to other UNIX versions. And because it is a high level implementation it can be added to a system without any tedious reconfiguration of the UNIX system. Under the usual conditions of a few, long lasting windows, with the major terminal activity in the current window the performance is reasonable.

Still, on supermicros, cpu cycles are at a premium, and WINDX requires IO to pass through several pipes. All this piping can tax the cpu. Also, programs which access the terminal directly at the device level can interfere with the display manager. To remedy this a new, low level windowing system (Glass+ ?) is being implemented. In this case the window system on the host is part of the tty device driver. While more difficult to install because integration into the device driver is required, this implementation should be completely transparent to all programs. In addition it should be a good deal more efficient, since it eliminates the use of tasks cooperating via pipes. This approach is preferable even for Blit style terminals since it would allow transparent access to each virtual terminal.

3. Other Applications for Windows.

By allowing offscreen memory, both Blit and GRAPHOS make pop up menu systems relatively easy to implement via

windows. An example of this kind of user interface is the IDraw program. IDraw is a program used to doodle on GRAPHOS. The screen is divided into several areas, each presented by a different window. One window is used to display one of several previously constructed menus, the balance being store off screen available for instant recall. A second window displays the current drawing options, while a third presents the easel for drawing.

The menus are drawn on the same page and are accessed by moving the menu window to the appropriate part of the page. The easel has its own coordinate system, independent of the surrounding screen windows, so the program does not need to worry about offsets to avoid menu areas. Instead it simply selects the easel page and sends normal drawing primitives. This type of coordinated window interface can be used to create very easy to use systems for presentation graphics and similar office automation tasks.

For this application windows have been used quite differently than in WINDX. IDraw uses windows to organize the screen in a modular fashion. Each window has a specific function : displaying selection menus, displaying current options, and displaying the drawing in progress. The subroutine responsible for each window can manage its own screen area without regard for other areas of the screen. This keeps the code clean and easy to deal with.

This modular screen management becomes very significant in environments where many processes need to manage part of the display. For instance, in chemical process monitoring we may have two tasks, one monitoring temperature, the other monitoring pressure. Each needs to display a graph of the recent state of affairs, but in vastly different scales and styles. Assigning a window to each task quickly solves the problem of screen management. It also allows us to change the screen layout without effecting the tasks. Windows can provide an efficient mechanism for screen management. In the same way structured programming organizes the programming job, windows can organize the screen management job.

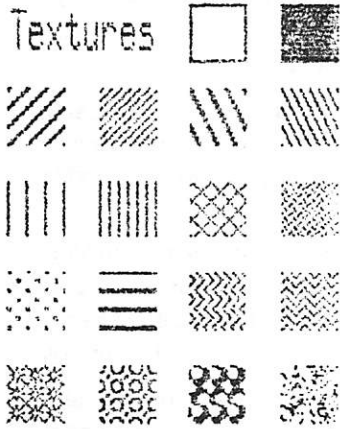
4. Conclusion.

After examining several approaches to window implementation, it is clear that the most important decision is how to distribute the window software between host and terminal. For an efficient system the terminal must be responsible for most of the window management. Expecting such capabilities in a terminal is quite resonable today, since most terminals are designed around some kind of microprocessor. Given such an intelligent terminal both high and low level implementations are acceptable.

References

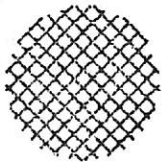
1. Norman Meyrowitz and Margaret Moser, "BRUWIN: An Adaptable Design Strategy for Window Manager/Virtual Terminal Systems," Proc. 8th Symp. Operating Systems Principles Vol. 15(5), pp.180-189, ACM (Dec. 1981).
2. Rob Pike, "Graphics in Overlapping Bitmap Layers," Computer Graphics Vol. 17(3), pp.331-356, ACM (July 1983).
3. Ithaca Intersystems, Inc., GRAPHOS Command Reference Manual, Oct. 1983.

Textures

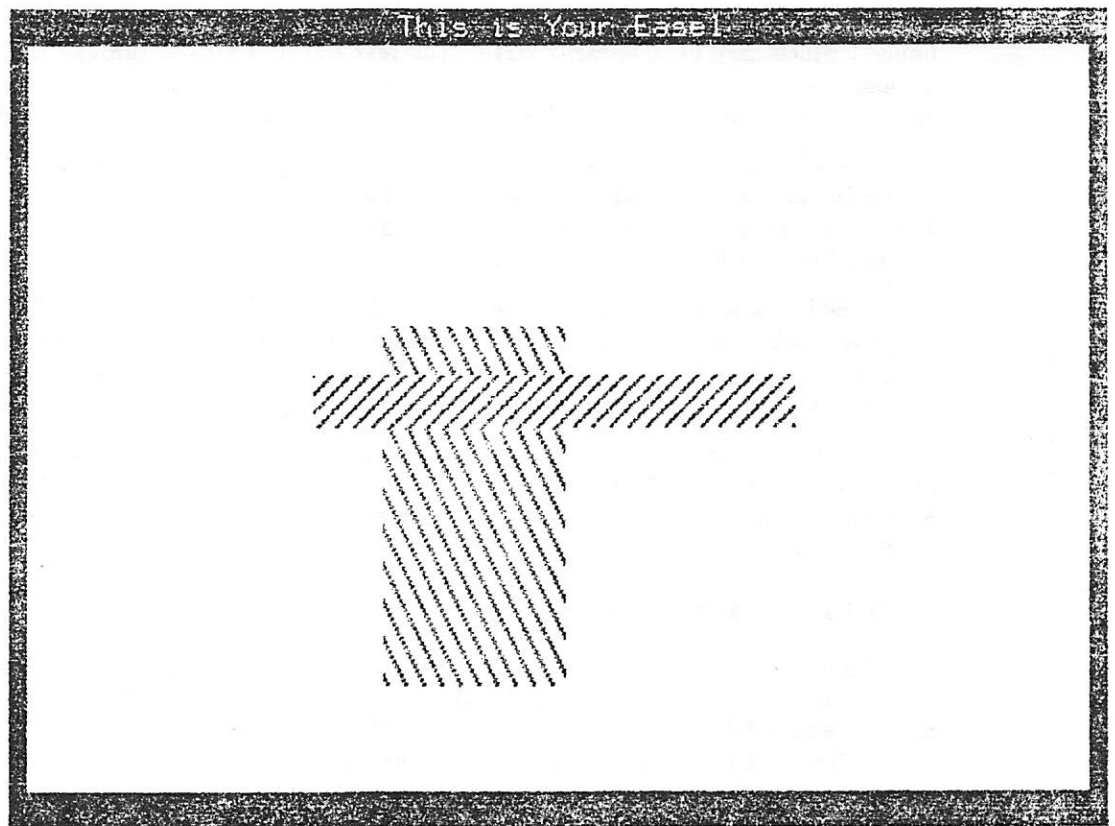


COLORS

RETURN



CURRENT OBJECT

GRID
ONORTHOGONAL
ONFREEHAND
ON

CIRCLE: Specify center and any circumference point.

GRAPHOS Application - Drawing Program

Ithaca InterSystems, Inc. 200 East Buffalo St. Ithaca, New York 14850

The Maryland Window System

Chris Torek

Mark Weiser

1. Introduction and Terminology

In the Maryland Window System, we have defined a window as a rectangular region on a physical terminal screen. Windows are constrained to remain entirely on the screen, although they may be covered by other windows. Each window is a completely independent entity (with the exception of "linked windows", which will be explained later).

The restriction to a physical screen size proved to be undesirable, so we introduced the notion of a "text buffer", which may be arbitrarily large. The text buffer is viewed through the "window buffer", much as a room may be viewed through a glass window. However, in Maryland Windows, the "glass" can be slid around over the buffer. This provides a means of displaying arbitrary amounts of text.

A window is usually drawn with a border or "frame" around it, so as to delimit the space available within it. In the Maryland Window System, the frames are optional, and are implemented with a more general mechanism which we call *margin settings*. The space allocated to these may also be used to provide special effects, such as highlighting a line in a menu.

In addition to an arbitrary number of windows, we allow a single "box" to be drawn on the display. The box is a hollow rectangular region of inverse video characters, and may be an arbitrary size, so long as it remains within the physical terminal screen. Thus the box may be used as another kind of window frame, for use when creating or moving a window, or as a roving block cursor, for selecting among several windows.

The window as a whole is what we refer to as a *window*. The text buffer is known as the *textbuf*, and the frame and special effects area is called the *winbuf*. The box is simply "the box", and in the current implementation, only one terminal screen may be controlled. There are no special provisions for keyboard input.

2. What's in a Window?

Windows have two kinds of text, three kinds of cursors, margins, and loads of operations. There can be text on the window itself, and text in the buffer onto which the window is looking. The window itself has margins which delimit normal cursor motion. The three cursors are for the window itself, the buffer under the window, and the window margin. All the usual terminal operations, plus a few new ones, apply to both kinds of text.

2.1. Cursors

Like a terminal, a window needs a cursor. The window's cursor shows where the next character will be placed within that window (though characters are actually written into the text buffer). The cursor is normally visible as a blinking block, though it may be turned off if desired.

Since a window actually has two pieces, the *winbuf* and the *textbuf*, it also has more than one cursor. The normal cursor (*window cursor*) is the one just described. There are two others: the *aux cursor* and the

buffer cursor. The aux cursor and the buffer cursor will be discussed later; for now it is enough to say that they are never visible, and that all three cursors are mutually independent.

2.2. Text

Characters within a window may be displayed with any combination of four modes: bold, underscored, blinking, and inverse video. Each window has a "mode" associated with it, which is the mode with which future printed characters will be displayed. To implement the frames and special effects, characters within the winbuf have an additional mode bit, to indicate whether or not they are "transparent". If a character in a window's winbuf is transparent, a character from the textbuf is displayed instead. In any case, the mode of the character from the winbuf will be used. This works as follows:

- 1) If the character from the winbuf is being displayed (e.g., for a frame), it is simply displayed with its proper mode.
- 2) However, if the winbuf character is transparent, the mode is exclusive or'ed with the mode of the character in the textbuf, and the result is used as the new mode for the displayed character.

Normally, the winbuf is filled with transparent blanks, but the aux cursor—or one of the special functions—may be used to write on the winbuf, to create special effects.

2.3. Linked Windows and the Buffer Cursor

Linked windows are windows which share the same text buffer (and the buffer cursor). This allows two windows to display the same text, or different portions of the same text. Linked windows can also be used to create arbitrarily-shaped windows, by careful placement on the screen and careful relative scrolling positions.

2.4. Text Buffers vs. Window Buffers

The separation between text buffers and window buffers introduces some complexity in operations performed on windows. More specifically, should an operation be performed on the winbuf, on the textbuf, or both? We chose to implement operations that affect the text buffer only, and operations that affect all of the visible portion¹ of the window (i.e., the entire winbuf, and the visible part of the window's textbuf).

2.4.1. Margin Settings and the Aux Cursor

Window buffers also have margin settings. These are mainly used internally to keep the window cursor from overwriting the frames (if any), but can be set if desired. Anything outside the margins is strictly off-limits; the space inside the margins is the window glass. However, the *aux cursor* is allowed to roam over the entire winbuf. It can be used to print characters directly into the winbuf at any position, either transparently or not.

Frames are thus created by writing non-transparent "+"s, "-"s, and "|"s—or, if the terminal supports them, special graphics characters—into the winbuf, outside margins that limit the inner area by one character on each side. "Tinted glass" may be created by writing transparent characters with special modes, exposing the character in the text buffer but modifying its display mode.

Text buffers, on the other hand, do not have margin settings. No part of the text buffer is protected. This leads to a problem: quite often, some of the text buffer is not visible. In a typical framed window, the entire top and bottom lines of the buffer, and the left and right edges, are not visible, since they are covered by the frame characters. Since the aux cursor could at any time erase part of the frames, we chose

¹ "Visible portions" can include covered or hidden portions. "Covered" is used to describe windows or characters that are obscured by other windows. "Hidden" is used to windows not displayed at all by specific request. The visible portion of a window's textbuf is the area that is inside the window's margins; this is irrespective of whether the window is hidden and/or covered.

not to allow the text buffer to be "moved down and right". This appears to have been a mistake, and may be changed in future versions of the Maryland Window System. (See Figure 1 for a display of characters obscured by frames.)

2.4.2. Operations

We have termed our two classes of operations "buffer operations" and "window operations". In general, window operations deal with the visible portion of the text buffer, and in many cases with the window buffer (but only the area inside the margins). Buffer operations are not limited to the visible text. Figure 1 shows what happens when a line delete is performed both as a window operation and as a buffer operation on a window that is smaller than its buffer. (If the window is the same size as the buffer, the difference between the two types of operations can usually be ignored.)

Window	Buffer	Window	Buffer	
+-----+	0123456789	+-----+	0123456789	
CDEFGH	ABCDEFGHIJ	CDEFGH	ABCDEFGHIJ	
MNOPQR	KLMNOPQRST	MNOPQR	KLMNOPQRST	(Before)
WXYZab	UVWXYZabcd	WXYZab	UVWXYZabcd	
+-----+	efghijklmn	+-----+	efghijklmn	
(delete line from window)		(delete line from buffer)		
+-----+	0123456789	+-----+	0123456789	
CDEFGH	ABCDEFGHIJ	CDEFGH	ABCDEFGHIJ	
WXYZab	KLWXYZabST	WXYZab	UVWXYZabcd	(After)
	UV cd	ghijkl	efghijklmn	
+-----+	efghijklmn	+-----+		

Figure 1: Windows and Buffers: Line Delete

When the cursor position is relevant to the operation, window operations are done at the window cursor position, and buffer operations at the buffer cursor. Only two operations (aux print and aux read) are done at the aux cursor position.

2.5. Scrolling

We have already mentioned that scrolling is performed by moving the window over the text buffer beneath. This is not precisely true. The window can be scrolled up, down, left or right, over its text buffer, but only so long as there is additional space within the textbuf. Windows are not allowed to "hang out" past the edge of the text.

A more conventional definition of scrolling is moving the existing text upward or downward, discarding lines at the bottom or top to make room for new text. The Maryland Window System provides this by making an exception to the scrolling limits. If the window is scrolled past the end of the buffer, the buffer is "extended" one line at the bottom, and a line is "erased" from the top, making the window fit once again. (The actual algorithm is to copy the text upward, keeping the window at the bottom of the buffer.) A similar exception is made for scrolling past the beginning of the buffer. These actions are known as "scrolling the buffer", and may also be performed without moving the window at all.

Windows will automatically scroll when necessary. When this happens, the number of lines scrolled is controlled by the window's *popup*. Popup defaults to one, but can be set anywhere from zero to the size of the window. Scrolling by n lines ($n > 1$) is exactly equivalent to scrolling by one line n times. Scrolling by zero lines is a special case: it is not scrolling at all. Rather, the cursor is brought to the top of the window

and the top line is cleared. Each time the cursor advances to the next line, that line is also cleared. When the cursor reaches the bottom of the window, the process is repeated. This is much faster on some terminals, but tends to be confusing.

2.6. Printing Text

There are three ways to print text (one for each cursor). Printing at the window and buffer cursors is generally more useful, and has less "drastic" effects: the aux cursor may be anywhere within the winbuf, and characters printed with it are required to have a mode. This mode includes (in addition to the standard four mode bits) the transparency bit. The character being printed is not interpreted in any way. If the character is not a displayable one (ASCII codes 32 to 126 inclusive), the results are terminal-dependent. Some terminals use special codes for the window frame characters, and look for such codes; others do not and may end up clearing the screen, repositioning the cursor, or doing other undesirable things. Therefore, the aux cursor must be used very carefully.

Both the window cursor and the buffer cursor check the character being printed, and take special actions for unprintable characters. Most of these are ignored, but the following are not:

BEL ASCII 7, or bell, rings the terminal's bell. If desired, this may be set to flash the terminal's screen instead, providing that the terminal has this capability. Visible bells are often nicer in a crowded working environment.

BS ASCII 8, or backspace, moves the cursor backwards one character. If the cursor is at its leftmost permitted column, it does not move.

TAB ASCII 9, tab, moves the cursor forward one tab stop. Tab stops are set at every eight columns. If there are no remaining tab stops, the cursor is moved to the rightmost column.

LF ASCII 10, linefeed, moves the cursor down one line without changing its column. (If CR/LF mapping is turned on, it moves it to the leftmost column.) If the cursor is on the bottom line, the window (or buffer) is scrolled.

CR ASCII 13, carriage return, moves the cursor to the leftmost column. (If CR/LF mapping is turned on, it also performs a linefeed.)

For the window and buffer cursors, autowrap and CR/LF mapping can be turned on and off. Autowrap means that when a character is written in the last column of the window or buffer, it is wrapped to the next line, as if a carriage return and line feed had been printed. This mode defaults on. CR/LF mapping makes both carriage return and linefeed go to the first column of the next line; i.e., either code performs the actions normally done by both. This mode defaults off. The aux cursor always wraps, and, upon reaching the last column of the last line, wraps to the first column of the first line. Since it does not translate special codes, it ignores CR/LF mapping. It should not normally be used to print CRs or LFs.

3. Update Optimization

The Maryland Window System keeps track of what the terminal's screen looks like, and when asked to update (*refresh*) the screen, attempts to do it with the minimal possible I/O. If a section of lines on the current screen matches a section on the desired screen, line insert and delete operations may be done to move the old lines to their new position. If subsections of an old line match those of a new one, insert and delete character may be used. Any number of arbitrarily complex operations may be done between screen refreshes; the update is computed at refresh time.

4. Applications

4.1. Window Shell

A windowing shell called *CWSH* has been built on top of the Maryland Window System. It uses multiplexed files under Berkeley 4.1 Unix² to fully emulate a terminal within each window. The unmodified cshell with full job control, unmodified vi, etc., can run in *CWSH* windows. The window shell has user-controllable bindings of keys to window commands, and also permits programs to control windows via "ioctl" calls. *CWSH* is approximately 3000 lines of C. More details are available in a separate paper.

4.2. Menu Library

One of the most common uses of a window system is to provide a menu-driven user interface to some application. A package of subroutines for creating and interfacing to menus was written on top of the Maryland Window System. The subroutine package handles input for selecting menus and selecting menu items. It saves the application program from any detailed knowledge about menus, returning information only about the user's action but not how it was made. Creating a menu-driven system is considerably eased by this package of routines. The menu library is about 1000 lines of C.

4.3. Lisp Interface

The lisp interface package provides the lisp programmer access to the Maryland Window System. Just about the only changes required were mapping structures used by the window system into s-expressions. The interactivenss of lisp allowed for some additional capabilities, such as routines for automatically creating windows based on a variable-sized "box" on the screen.

The lisp interface package exists in two forms. One is the usual set of functions which map directly into Maryland Window functions, and the other is as a "windows" abstract datatype under the Flavors object-oriented programming facility of Franz Lisp. The advantage of a flavors interface is that the "windows" abstract object can be mixed with other abstract objects to create more complex structures (e.g. mixing a windows flavor with a linked list flavor would create an abstract object for representing linked lists of windows). Originally only a flavors interface existed, but it was found that in many cases the windows flavor was not being mixed with anything so that a needless overhead in storage and function indirection was occurring. Thus the window functions were made available directly. However, for more complicated structures built on top of windows, such as menus, the flavor is beneficial and is still used.

4.4. TEXTNET

TEXTNET is a menu driven system for creating, browsing, and critiquing text Documents are organized hierarchically, with Table-of-Contents (TOC) nodes on the interior and Chunk nodes at the leaves. Chunk nodes correspond to indivisible pieces of text while TOCs capture hierarchical structures corresponding (loosely) to chapters, sections, etc. User interaction is entirely through windows for menus and information display.

TEXTNET is continually showing windows with menus in them from which the user must select an action. Menus are used to create, browse, and critique documents. TEXTNET also supplies global commands which can be used at any time to create a new window context. One example is the "S" command, which shows a menu corresponding to the table of contents for the current document from which the user can move anywhere in the document. Others are "?", that moves the user to a TEXTNET help menu, and "C", that allows the user to critique whatever it is they are currently reading.

TEXTNET is approximately 5400 lines of lisp code and uses 5 different kinds of windows and 15 different kinds of menus. There are now 220 thousand characters in TEXTNET contained in 9 documents, and this figure is growing rapidly.

² Unix is a trademark of Bell Laboratories

4.5. Afloat

Afloat is a language independent debugging tool which allows a programmer to discard irrelevant details and focus on her problem in a natural way. It has a menu driven command interface based on Maryland Windows. Windows are used to display program text, dataflow information about a program, and suggestions about the location of bugs. Afloat is about 3330 lines of C, and uses 4 different kinds of windows.

5. Future Enhancements

We plan to extend Maryland Windows to support multiple screens. An internal change which should greatly increase speed on systems without floating point is in the works. We are considering ways to provide better control of input, including function key mapping, so that programs can present a more uniform user interface. Also in the "user interface" department is the idea of "extensible buffers": upon reaching the bottom or right of a text buffer, rather than discarding text, the textbuf could be expanded. This would allow the user to review previous input or output, or even create a linked window to keep information on the screen permanently.

6. Current Status

The basic Maryland Window System subroutine package is approximately 7000 lines of C, and runs under Berkeley 4.1 and 4.2 Unix. It has been in use at the University of Maryland Computer Science Department for more than 2 years, and is available for distribution. An early version came out on net.sources. Later versions have been distributed on tape to more than 70 university and industrial sites, some of whom are known to be using it. Contact Diane Miller (despina@maryland, {seismo,allegro}!umcp-cs!despina, (301) 454-7690) for distribution information.

Appendix : Alphabetical List of Functions

Functions marked with "•" are actually macros.

- Ding()
- Max(a,b)
- Min(a,b)
- WAcursor(w,row,col)
- WAread(w,charonly)
- WBclear(w,howto)
- WBclearline(w,howto)
- WBcursor(w,row,col)
- WBdelchars(w,n)
- WBdelcols(w,n)
- WBdellines(w,n)
- WBinschars(w,n)
- WBinscols(w,n)
- WBinslines(w,n)
- WBputc(c,w)
- WBputs(s,w)
- WBread(w,charonly)
- WBscroll(w,rows)
- WBtoWcursor(w)
- WCHAROF(c)
- WMODEOF(c)
- WWcursor(w,row,col)
- WWtoBcursor(w)
- Waputc(c,m,w)
- Waputs(s,m,w)
- Wauxcursor(w,row,col)
- Wback(w)
- Wborder(w,ulc,top,urc,left,right,llc,bottom,lrc)
- Wbox(xorg,yorg,xext,yext)
- Wboxfind()
- Wcleanup()
- Wclear(w,howto)
- Wclearline(w,howto)
- Wclose(w)
- Wcloseall()
- Wcurdown(w,n)
- Wcurleft(w,n)
- Wcurright(w,n)
- Wcurup(w,n)
- Wdelchars(w,n)
- Wdelcols(w,n)
- Wdellines(w,n)
- Wexit(code)
- Wfiledump(file)
- Wfind(id)
- Wframe(w)
- Wfront(w)
- Wgetframe(ulc,top,urc,left,right,llc,bottom,lrc)
- Whide(w)
- Winit(settings,nomagic)
- Winschars(w,n)
- Winscols(w,n)
- Winslines(w,n)
- Wlabel(w,label,bottom,center)
- Wlink(ew,id,xorg,yorg,xext,yext,bcols,brows)
- Wmove(w,xorg,yorg)
- Wnewline(w,on)
- Woncursor(w,on)
- Wopen(id,xorg,yorg,xext,yext,bcols,brows)
- Wputc(c,w)
- Wputs(s,w)
- Wread(w,charonly,winonly)
- Wrefresh(SkipInputCheck)
- Wrelscroll(w,rows,cols,cwin)
- Wretroline(w,m,inwin)
- Wscreenize(rows,cols)
- Wscroll(w)
- Wsetbuf(w,bcols,brows)
- Wsetframe(ulc,top,urc,left,right,llc,bottom,lrc)
- Wsetmargins(w,inxorg,inyorg,inxext,inyext)
- Wsetmode(w,m)
- Wsetpopup(w,p)
- Wsize(w,xext,yext)
- Wsuspend()
- Wunhide(w)
- Wwrap(w,on)
- bcopy(from,to,count)

A Text-Oriented Terminal Multiplexor for Blits

Rob Pike
Bell Labs 2C524
Murray Hill NJ 07974
201 582-7854
research!rob

Abstract

The Blit is an experimental bitmap terminal built specifically for interactive graphics on UNIX systems. Recently, Teletype has announced the DMD 5620, a product based on the original Blit terminal.

The research software for the Blit terminal has been rewritten to provide character-level processing of text on the display. This allows the user to edit and resend previous commands, edit and modify the output of one command and apply it as input to another, copy text from window to window, and so on. This software, called mux (multiplexor), is interesting for several reasons:

- * The implementation, built using the stream I/O in the UNIX Eighth Edition system, places the terminal driver in the terminal itself, rather than in the operating system kernel. Superficially similar programs for other terminals have increased the overhead on the host computer; mux reduces it. For example, typing echo is provided directly by the part of mux running in the terminal, and characters are sent to the host a line at a time. Nonetheless, mux provides a real Unix terminal driver, with the standard ioctl interface.
- * The popular 'CRT' style of terminal driver is obviated by the general character-level editing of the system.
- * The interactive capabilities of mux surpass those of most window systems because of the (transparent) coupling of editing, window management and terminal services in one program. At the same time, because mux operates on plain text -- a byte stream -- it is completely general.
- * Finally, and this is perhaps most important, having a terminal that provides high-quality interaction for all programs allows the attention to interactive issues to be focused in one place. Features such as shell history and reply commands in mail are superfluous (and little-used) given mux. For example, history is available to any program running under mux, without prior arrangement by the program. However, such features are made obsolete only because the graphical interface mux provides is comfortable and effective enough that users are content with mux.

This talk will discuss these points and others.

A Multiprocessor UNIX* System

Maurice J. Bach
Steven J. Buroff

AT&T Bell Laboratories
190 River Rd.
Summit, NJ. 07901

This paper¹ will describe the development of a UNIX operating system that runs on multiprocessor configurations. The system currently runs on AT&T 3B20 computers, but the ensuing discussion applies equally well to other machine architectures that support a multiprocessor environment. Existing user level C programs can run without recompilation on uniprocessor and multiprocessor models of the 3B20 computer, unless the program contains system dependent code (e.g. the command to determine process status, *ps*). The terminal interface, file system format, process hierarchy, and all other user visible aspects of the operating system appear identical to those on a single processor UNIX system.

1. *Why Multiprocessor Systems*

UNIX systems are commonly used for software development, where programmers working on a project must communicate and share data with each other. But many software development projects, although they start out small, later outgrow their original computing capacity, so that a single computer no longer adequately supports all users.

A multiprocessor capability allows a smooth growth path for projects that can start small with a single processor and, as their computing requirements expand, can add more processors to form a larger, more powerful system. Such growth is usually less expensive and less disruptive to end users than acquiring a new and larger machine.

Another advantage of a multiprocessor system is that it is potentially more robust. If a hardware failure results in one processor becoming inoperable, the system can potentially recover from the problem. The users would not have to take any special action and would not notice any difference in system services except reduced performance. It is still an open problem how to diagnose and fix such problems on a multiprocessor UNIX system while the system is active, so the systems described here require a system reboot to restore operation. However, they execute in single processor mode so that failure of one processor does not prohibit booting and running the system on the other processors.

2. *Kernel Changes Required For a Multiprocessor Unix System*

The UNIX system was originally developed to run on a single processor, and the code assumes that the kernel is never preempted except for processing of interrupts. Hence, kernel data structures do not need to be protected unless referenced by an interrupt routine, and if so, the data can be protected by locking out interrupts. This is normally done by raising the processor priority level high enough to prevent the type of interrupt from occurring.

In the multiprocessor systems described in this paper, however, raising the processor execution level does not prevent corruption of system data structures, as all processors can simultaneously execute kernel code. It is possible to solve the problem by preventing two processors from simultaneously executing kernel code. When a user makes a system call, the system would make sure that no other

* UNIX is a trademark of AT&T Bell Laboratories

1. This is a condensed version of a paper which will appear in a future issue of the *Bell System Technical Journal*.

processor is currently executing in the kernel. But benchmark programs show that UNIX systems typically spend between 40% and 50% of their time executing in the kernel, so this solution does not achieve the full performance potential of the hardware, nor does it scale well to configurations of more than two processors. Instead, the solution presented here allows many processors to execute kernel code simultaneously by inserting Dijkstra-style semaphore operators around critical regions of code.

3. Semaphores

A semaphore² is an integer valued data structure on which the following restricted set of operations can be performed.

<i>init</i>	Initialize the semaphore to an integer value.
<i>psema</i>	Decrement the value of the semaphore. If the resulting value is less than zero, then suspend the executing process and place it on a linked list of processes sleeping on the semaphore. When awakened, the process priority is set to the value supplied as one of the parameters to <i>psema</i> . If signals are pending against an awakened process, the value of the priority parameter determines whether they are deferred or caught.
<i>vsema</i>	Increment the value of the semaphore. If the resulting value is less than or equal to zero, then awaken a process that suspended itself doing a <i>psema</i> on the semaphore.
<i>cpsema</i>	If the value of the semaphore is greater than zero, then decrement it and return true. Otherwise, leave the semaphore unmodified and return false.

Semaphore operations are atomic. That is, if two or more processes try to do operations on the same semaphore, one completes the entire operation before the others begin.

To protect a particular resource such as a table or linked list, a semaphore is associated with that resource and typically initialized to 1 when the system is booted. When a process wants to gain exclusive use of the resource, it does a *psema* on the semaphore, decrementing the semaphore value to zero (assuming it was one) but allowing the process to proceed. The process now has exclusive use of the resource. If other processes attempt to gain control of the resource, their *psemas* will decrement the semaphore value and will suspend process execution. If the value of a semaphore is negative, then its absolute value is equal to the number of processes that are suspended waiting for that resource. When the process that has control of the resource relinquishes it, it does a *vsema* on the semaphore, releasing the semaphore and awakening a suspended process, if any. The awakened process is now eligible for scheduling when a processor becomes available according to the usual process scheduling algorithm. When scheduled, the awakened process returns from the *psema* call without knowing that it was temporarily suspended, and when it finishes with the resource, it should do a *vsema* to release the semaphore and to awaken the next waiting process, if any.

Besides protecting a shared data structure, semaphores can also be used to wait for an event. In this case, the semaphore is initialized to zero. Processes awaiting the event do a *psema* to suspend themselves until the event occurs, and processes recognizing the event do a *vsema* to awaken sleeping processes. Semaphores can also be used to count resources. A semaphore that is used to count the number of resources in the system is initialized to the appropriate number. When the resource is allocated, the *psema* decrements the semaphore value, and when the resource is freed, the *vsema* increments the semaphore value, so that it always conforms to the number of available

2. The semaphores being described here are a strictly internal mechanism and have nothing to do with the user level interprocess communication facility of the same name that is described in [1].

resources. If the number of available resources drops to 0, processes will sleep in the *psema* until another process releases a resource and does a *vsema*.

The *cpsema* operation is used to lock a resource only if it is immediately available, and other action besides sleeping is to be taken if the semaphore is unavailable. This is used for deadlock prevention.

Single processor UNIX systems use the *sleepwakeup* mechanism for process synchronization to voluntarily suspend and resume execution waiting for an event to occur. When a single processor system does a *wakeup* call on a resource, all processes *sleeping* for that resource are awakened. Often the resource must be used exclusively, so all but one of the awakened processes will test the resource, find it busy, and again go to *sleep*. In multiprocessor systems on the other hand, it is undesirable to awaken all sleeping processes because all such processes could not assume exclusive access to system structures. So a *vsema* only awakens a single process that will in turn awaken another sleeping process. A process that executes a *psema* knows that it has control of the resource and will not fall asleep again waiting for the resource to become "ready."

The kernel of the multiprocessor systems has been modified to account for the change in semantics of sleeping. Calls to the *psema* and *vsema* functions replace calls to the old *sleep* and *wakeup* functions as there is one set of process synchronization primitives (semaphores) instead of two.

4. Scheduling

The UNIX system scheduling algorithm remains identical to that of uniprocessor systems. The system contains one set of runnable processes, and it schedules them to run as processors become available. A new process state indicates that the process is currently running, so that other processors will not schedule that process again.

5. Drivers

In principle, there is no difference between device drivers and other parts of the operating system as far as conversion for running on a multiprocessor is concerned. Data structures must be locked, *sleepwakeup* calls must be replaced by *psema/vsema* calls, and special consideration must be given to interrupt routines as described previously.

But more than half of the UNIX operating system currently consists of device drivers, and new drivers are being added at an accelerating rate to support new peripherals and to provide new or enhanced services. In practice therefore, the number and volatility of the drivers make it difficult to change them for multiprocessor systems and keep them up to date with changes made for other UNIX systems, so it is important to keep most driver code identical over all implementations. Three changes had to be made to the system to allow this.

First, drivers are locked before they are called. Driver calls are table driven via the *bdevsw* and *cdevsw* tables, and the drivers are locked and unlocked around the driver calls using driver semaphores added to the tables. Various methods of driver protection are encoded based on system configuration. The levels of protection vary from no protection (protection is then hard coded in the driver), to forcing the process to run on a particular processor (useful in AP configurations where only one processor can do the I/O), to locking per major or per minor device type. Each call to a driver routine is now preceded by a call to a driver lock routine and followed by a call to a driver unlock routine.

The second change was to reimplement *sleep* and *wakeup* subroutines that could be called by device drivers without changing the original driver code. Since the old UNIX operating system *sleep* routine uses arbitrary addresses in memory to sleep on, the new routines use hash lists of semaphores to actually suspend the process, and the address being slept on (a *sleep* parameter) is stored in the process table. Since a semaphore already heads a linked list of all processes suspended on the semaphore, the wakeup routine has only to search this list to find all processes to awaken. The *sleep* routine unlocks the driver semaphore so that other processes can access the driver while the original process sleeps, and it relocks the semaphore when it awakens from the sleep. The *sleep*

and *wakeup* routines are intended to be used only from drivers. The main kernel code still uses *psema* and *vsema* directly.

In addition to the locking before calling driver routines, locking must also take place when handling interrupts, since the interrupt is no longer blocked by raising the processor execution level. Before the device interrupt handler is invoked, the semaphore for the device (if any) is locked via *cpsema*. If the lock succeeds, the interrupt gets handled; if the lock fails, the interrupt is queued but not handled immediately. When the process that currently has the semaphore locked is finished with the semaphore, it handles queued interrupt requests.

6. *Performance*

The system can be configured for execution in uniprocessor or multiprocessor environments. Running the system in a two processor configuration results in a system throughput increase of 70% over the range of general workloads. The "uniprocessor mode" of the system executes just as fast as the pure uniprocessor system from which it was derived. Running a system configured as a multiprocessor on one processor suffers less than 5% performance degradation due to the overhead of semaphore operations.

REFERENCES

1. *UNIX System Users Manual*, Release 5.0, June 1982, Bell Laboratories Inc.

A Demand Paging Virtual Memory Manager for System V

 Richard Miller
 Human Computing Resources Corporation
 10 St. Mary Street
 Toronto, Canada M4Y 1P9
 (416) 922-1937
 {decvax,utzoo}!hcr!miller

Introduction

This paper describes work done at Human Computing Resources Corp. during the last quarter of 1983, to incorporate a demand-paging memory management scheme into UNIX System V.

The goal of the project was a modest one: to make the smallest possible change to the standard AT&T kernel in order to provide a virtual memory environment for user processes. The functionality of UNIX was to remain identical at the system-call level: no effect of the change should be visible outside the kernel, and no modifications required to commands or user programs.

Although it is tempting to use virtual memory as a basis for extending UNIX with a multitude of new features borrowed from more elaborate operating systems -- such as dynamic linking, shared subroutine libraries, and memory-mapped files -- even a "transparent" virtual memory facility can have some important benefits. The most obvious is that program size is no longer limited by available physical memory, but only by the logical addressing range of the processor and memory management hardware. Also, by loading only those pages of each process which are being actively referenced, more processes can be resident at the same time, resulting in a more effective use of both CPU and memory resources.

Of course, these advantages have their tradeoffs. While a large address space may be essential in some applications, all too often the illusion of limitless free memory leads to a cancerous growth in program complexity and consumption of real resources. As an example, consider the humble `cat(1)` command, which in moving from a real to a virtual environment (V7 to 4.1BSD) acquired six option flags and more than doubled in size. And even the best paging algorithm will not permit the population of processes to grow without bounds. At best, we can accommodate a few more users before "thrashing" begins. At worst -- with small processes and plenty of real memory -- the overhead of page faults and fragmented swap space will make performance poorer.

An important aim of our paging design was to limit the potential performance cost: processes small enough not to require virtual memory should run at least as efficiently as on the original swapping system.

Finally, the new memory manager was intended to be as machine independent as possible. The original implementation was for a MC68010-based machine, with a memory management unit functionally very similar to the one used in the Sun workstation; the design should be easily adaptable to any hardware architecture able to support demand paging.

Design of the Memory Manager

System V, like its ancestors, does not have a "memory manager" as an identifiable component of the kernel. Rather, a number of functions in separate source files are responsible for manipulating memory at different points in the lifetime of a process. Implementing virtual memory involves changing each of these functions according to the principle of demand paging: pages in the address space of a process are assigned to real memory locations only "on demand" -- when they are first referenced by the process.

Thus, when a program is initiated by the `exec(2)` system call, the virtual memory manager does not actually load the new process image into memory. Page tables are set up to reflect the new sizes of text, data and stack; but entries are marked as "invalid" and no real memory is allocated. When the process begins execution, its first instruction fetch will cause an addressing fault -- because it hasn't been loaded yet. This trap will be caught by the page fault handler -- the major new component of the paging kernel -- which then allocates a page of memory, reads in the contents from the program file, updates the page table to make the reference valid, and returns control to the process to restart the instruction. Similarly, the first data reference will generate another addressing fault, causing the page fault handler to allocate a new memory page. If the referenced page is in the initialized data segment, it will be read from the file; if in uninitialized data (`bss`) or stack, it needs only to be cleared to zeroes. In this way the program gradually loads itself in.

When the address space of a process expands -- whether from an explicit `brk(2)` request or an automatic extension of the stack -- the page tables for the new area are marked "invalid"; real pages will be allocated and cleared later by the page fault handler.

The principle of deferring allocation of real memory can be applied to the `fork(2)` operation as well, by the technique of "copy-on-write". Instead of copying data and stack segments, the child process can temporarily share the pages of the parent. The page tables of both parent and child must be changed to make the shared pages write-protected, so that an attempt by either process to modify the shared memory will cause a trap, allowing the page fault handler to duplicate the referenced page. Since a fork is usually followed quickly by execution of a new program, it is likely that only a few pages will ever actually need to be copied.

The initial version of our paging system does not use copy-on-write, for a pragmatic reason: the hardware used in the first implementation forces an extremely large page size of 2048 bytes. This size is only

slightly smaller than the entire data segment of the standard UNIX shell, which is responsible for the great majority of fork operations; so that trying to copy the data one page at a time would not make much difference.

The Page Fault Handler

The operation of the page fault handler is quite straightforward. It is invoked from the trap handler when an "invalid page" addressing fault occurs, and runs synchronously with the user process, much like a system call. Its first job is to examine the page table entry associated with the virtual address being referenced, to determine where to find the contents of the missing page. On some hardware architectures, there may be enough spare bits in the page table entry to encode this information; on other machines, we must provide a separate software table which parallels the hardware page table.

The simplest case is a "zero-fill" page, resulting from the first reference to a page of bss, or data newly allocated by brk(2). When a real memory page has been allocated, it is simply filled with zeroes.

When an initialized data or text page is first referenced, it must be read in from the a.out file -- sometimes referred to as "demand loading". The reading is done using the normal buffered I/O mechanism of the kernel. This means that -- in contrast to 4.1BSD, which uses direct "raw" I/O for demand loading -- the filesystem block size is independent of the memory page size, and no special a.out format is required. A complication to our approach is that the buffered I/O routines are not reentrant, since they use fixed locations in the "U area" to record the address and size of the process buffer. Since a page fault can occur during a user I/O operation which references an invalid page, this information must be saved and restored by the page fault handler.

The page fault handler may also find missing pages in the swap area, when a program has been partially or completely swapped out because of competition for main memory. In this case, raw I/O will be used to load the pages. To avoid the high performance penalty of single-page transfers to and from the swap device, whenever possible the I/O will be done in larger "clusters". When a page must be read in, a number of contiguous pages can be fetched at the same time, so that if they are referenced soon they will already be found in main memory. Up to a full track can be read without much more cost than reading one page, since the seek and rotational delay times of a moving-head disk are typically much greater than the transfer time.

Another way to reduce the cost of paging I/O is to "reclaim" free memory pages whose contents are still valid. This can be simply done without lengthy searches or hash tables, by viewing the pool of free pages as a "cache" for the swap device. A table with one element per page of real memory records the block number in the swap area where each page was last written; when a page is reallocated, the entry is cleared. Whenever a page is written to the swap area and freed, the

page table entry is marked "invalid", but the old memory page frame number is left intact. Thus, when the page fault handler needs to read back the page, it can directly look up the cache table entry indexed by the old page frame number -- if it still corresponds to the right swap block, the page can be reclaimed and no I/O is required.

The Working Set Scheduler

The demand paging mechanism determines when real memory is allocated to a process; we also need a "page replacement policy" to decide when memory pages are to be freed. Page replacement in our memory manager is based on Peter Denning's "working set" model.

Denning first proposed the idea of working set to describe the property of "locality" observed in the behaviour of programs: memory references are not uniformly distributed over time, but tend to be clustered in a limited subset of pages which changes with different "phases" of a program. Informally, the working set of a process is the set of pages which it is actively using at a given time. More precisely, we define the working set $W(k, t)$ at time t to be the set of all distinct pages referenced during the last k units of process "virtual time" (i.e. time during which the process was actually running). The parameter k is the working set "window size": the smaller the window, the more frequently a page must be referenced in order to remain in the working set.

A "working set scheduler" attempts to keep the entire working set of an active process resident in main memory. If a reasonable window size is chosen (and Denning has shown that performance remains quite stable over a large range of k), this should guarantee that the process can execute efficiently without excessive page faults.

When there is not enough real memory available to hold the working sets of all running processes, some process must be selected to be swapped completely out to disk, allowing the remaining processes to continue to run efficiently. Our memory manager uses the "standard" UNIX swap scheduler to perform this operation.

The advantage of working set scheduling is that it is a "local" policy: the memory requirements of each process are determined independently of the behaviour of other processes. In contrast, the global "clock" replacement policy used in 4.1BSD examines all memory pages cyclically, freeing those that have not been referenced recently in real time. This means that a process which is using more CPU time has more opportunity to reference its pages, and will "steal" pages from less compute-bound processes. Since the working set scheduler measures references in process virtual time, the coupling between CPU and memory scheduling will not occur.

The working set mechanism also provides a natural protection against thrashing, by swapping out processes automatically when memory is scarce. By comparison, without information about active working sets, the 4.1BSD memory manager has no way to recognize overcommitment of

main memory, and must use a separate load-control mechanism: imposing an absolute upper bound on the paging rate by limiting the number of pages being freed per second. The unfortunate consequences of this policy can be seen when a single process is bigger than physical memory (for example, running `fsck(1)` on a small workstation) -- the system can spend much of its time waiting for clock ticks instead of loading the pages it needs.

A rigorous implementation of the working set policy would require hardware to record the last reference time for each page frame. We implement a "sampled" approximation by performing a scan through the page tables of an active process after each window-sized interval of CPU time, examining and clearing the "referenced" bit for each resident page. Every page for which the "referenced" bit was set is considered to be still part of the working set; other pages are freed. Of course, this trimming of working sets is only performed when the supply of free memory drops below a given threshold.

In the case of machines with no hardware "referenced" bit in the page table (such as the VAX-11), it will be necessary to simulate the same effect, for example, by supplementing the "valid" bit by a software "really valid" bit. In place of clearing the "referenced" bit, we clear the "valid" bit, but leave the "really valid" bit set. A reference to the page causes a trap; but instead of invoking the full power of the page fault handler, the trap routine can determine that the "really valid" bit is on, set the "valid" bit, and return with a cost of only a few instructions.

Conclusions

By concentrating on the single implementation issue of demand paging, we have added a virtual memory capability to System V with a minimum of disturbance to the kernel. The only command which needed modification was `ps(1)`-- which can arguably be considered part of the kernel since it works by reading the kernel's private data areas.

The new memory management routines are reasonably portable: they were recently moved to the National Semiconductor 16032 in less than one week. Performance of the paging kernel seems similar to the original swapping system; more quantitative results should be available by the time this paper is presented at USENIX.

Acknowledgments

Thanks to Paul Neelands for inspiration and perspective.

Resource Controls, Privileges, and other MUSH.

Robert Elz

Department of Computer Science
University of Melbourne
ps + 2

ABSTRACT

MUSH is the collective name for a set of generally independent resource controls implemented at the University of Melbourne. This paper examines the system as it currently exists.

1. Introduction

At the University of Melbourne, in the Computer Science Department, we have, as do many other sites, the problem of far too many users competing for insufficient resources.

The ideal solution would be to simply purchase more of everything, but unfortunately, that solution is not within our means.

Our compromise solution is the set of resource controls described below, which aim to guarantee to each user, that he will receive at least some minimum quantity of the resources available.

There are three resources that are currently of concern: disc space, terminals, and cpu time.

In addition, given that the principal users of our system are undergraduate students, we need some method of controlling their access to each other's data. It turns out that the standard 4.2BSD UNIX[†] protection scheme is almost, but not quite, sufficient for our needs. We require (that is the rules of the department require) that undergraduates be prevented from giving away copies of their work to friends when doing subjects that specify that submitted work must be done individually.

[†] UNIX is a Trademark of Bell Laboratories.

One or two very minor changes permits us to enforce this, or at least make it considerably more difficult to overcome.

2. Disc Quotas

Little need be said here about the Melbourne disc quota system, as it has been distributed with 4.2BSD, and will be familiar to many readers. However, a brief mention is needed for those few unfortunates who have no access to that system.

Disc quotas provide both a block and inode limit, on a per filesystem basis. Users can be given limits on some filesystems, but open access on others. Limits can be set to 0, preventing any usage at all.

There is a *soft limit* or *quota* and a *hard limit*. Users are expected to remain under their *quota* when logged out, but may exceed that when connected. The hard limit is absolute, and may not be exceeded. Users who log out over quota are warned at their next login. Continued repetition of this warning causes the user's hard limit to be effectively set to zero, until he finally logs in under his quota. That is, all he can do is remove files.

The quota system is implemented in the kernel as that is the only place with access to the required information.

3. Sharing available CPU time

There are three types of limits that can be applied in the case of a need to share an overloaded cpu. Each of them has its advantages, and each fills a slightly different role.

3.1. Share Scheduler[†]

The object of a share scheduler is to allocate available cpu time among competing processes, at the microscopic level. Each user is given some number of 'shares' which represent his entitlement to the processor. Users with more shares should be given correspondingly more cpu time. Note, that only the shares of users currently using the processor are considered when performing these calculations.

[†] The share scheduler described here was originally implemented at the University of New South Wales by Andrew Hume. Implementation methods have changed, but the basic ideas and algorithms have not altered since that implementation.

In addition to the number of shares, which is almost a constant, a number representing the amount of resources that each user has consumed in the recent past is maintained. For two users with equal numbers of shares, the user with a lower usage will be entitled to more cpu time. Of course, should he make use of his entitlement, his usage will rise, eventually to a point where it equals, or exceeds, that of the other user.

The share scheduler is implemented as a clock driven routine in the kernel that re-evaluates each users entitlement every few seconds. It then adjusts the priority fields that the standard UNIX scheduler uses when selecting the next process to run.

The kernel increases a *cost* variable for each user whenever it performs any service for that user. There are a basic costs for the use of memory, and for cpu ticks, that are computed every second. In addition, system calls, page faults, i/o requests, etc, all contribute to the cost charged to the user.

The share scheduler adjusts each users' usage by an amount calculated from this cost, and a value representing the users average cost per period in the last few minutes. If the user's rate of use drops, then his usage figure will gradually drop too.

There is also user level process that interacts closely with the kernel scheduler. This is the MUSH daemon, and is a well known process, similar to */etc/init*. When the last process for a user exits, the kernel informs the MUSH daemon of the users final usage. When the first process for a user is started, by any means at all, the kernel informs the MUSH daemon and expects the users usage to be returned to its previous value. Actually, since the user has been incurring no costs during the entire period that he had no processes running, the returned usage will have been decreased by the same amount that it would have dropped had there been a process present incurring a zero cost.

The daemon maintains a file of usages, and other data that will be mentioned later. It communicates with the kernel, and other processes, using IPC. The IPC scheme currently used is a particularly horrid one created especially for the purpose.[‡]

[‡] MUSH predates 4.2BSD. Eventually, it is planned to convert MUSH to use the standard IPC mechanisms available in Berkeley UNIX.

3.2. Cpu time limits

A second method of controlling cpu usage, is to impose a maximum cpu time limit on each of the users' processes. This is principally useful in preventing accidental looping programs, especially those written and run by novice users. The 4.2BSD kernel already provides resource limits of this kind. Each process has several limits, for each of which there is a *current* and a *maximum* value. The process is free to alter the current value of any if its limits, but not so as to exceed the maximum.* Child processes inherit limits from their parents.

At Melbourne, we have made use of this facility, to restrict the available cpu time to user processes (and to restrict core file sizes, program sizes, etc). *Login*(1) and *su*(1) set the current and maximum values of the limits to a value found in a file indexed by the user's uid.

This works quite well for its intended purpose, though is not unbeatable. Eg: some users have discovered that ignoring cpu time limit signals effectively mean that the limit vanishes. This does not worry us unduly, as, these limits are primarily regarded as insurance against accidents.

3.3. CPU rationing.

A third possible method of controlling access to the cpu is to allocate users some number of cpu seconds, and to simply refuse them any further cpu time when their ration has been reduced to zero. The ration might be a fixed number, allocated when the account is created, or one that is increased each, hour, day, week, or whatever.

Currently, no such scheme is implemented in the MUSH system, though adding it would not be difficult. The kernel would merely need to deduct the cpu time used by each process from the remaining ration as it exits, and then set the resource limits for new processes to values no greater than the available ration when they commence. The MUSH daemon could provide a storage facility for inactive cpu rations in a similar manner to that used for inactive usages. Should the ration ever decrease to zero, the MUSH daemon could also handle this, assuming it is informed, in a manner similar to that it uses to handle terminal time limits (see next section). Should using the time of process exit prove to be too coarse a measurement, then the share scheduler could easily

* Root may increase maximums, naturally.

manage the cpu ration along with its other duties.

4. Terminal access limits

The system on which most of this development has been done has almost 1400 lines in its password file. Of these, at least 700 represent active users. There are 7 DZ's, 1 VMZ, and one local mux that supports 15 terminals attached to the 780. This gives a maximum number of terminals of 87. Several of these lines are connected to various printers, other machines, etc. Our actual maximum simultaneous login count is something just over 50. Clearly, a way to share the 50 terminals among the 700 competing users is needed.

One possible solution, adopted at the Universities of Sydney and New South Wales, would be to adopt a booking system, where users could book a terminal at a particular time. Then, he would be entitled to use that terminal during the period booked. This is a relatively simple and clean way to control terminal access, and it guarantees to the users that at some particular time there will be a terminal available for them.

At Melbourne, at least until very recently, most of our terminals have been connected through a port selector switch. There is no correspondence between terminals and tty numbers. This makes it hard, if not impossible, for a booking scheme to be implemented.

Our solution, is to ration terminal connect time. There are several distinct rations that can be applied simultaneously. First, is an absolute maximum number of minutes of login time that can be used. In practice we make very little use of this. Next, a user may be permitted to use only some number of minutes of terminal time each day, or week. There can also be a maximum login session length, no individual login session can last for longer than this, and there is an enforced gap between sessions. This limit is not enforced if there are free terminals. Users may also be prevented from logging in during certain periods of the day (or week).

While not ideal, this serves to adequately share terminals in our environment. It has the side effect, which in some quarters is regarded as more important than the primary goal, of preventing over-enthusiastic students from spending an excessive amount of time on the system.

4.1. Implementation

The kernel has been augmented to be able to distinguish between login processes, and others. A login process is one that is a legitimate child of `init`.[‡] The distinction is made by a flag bit in the process table, set by a system call.

Whenever such a process switches to a new user, or exits, a message is sent to the MUSH daemon, informing it of the change. Thus the MUSH daemon can track user logins and logouts. The daemon then manages the login limits, warning users when their time has almost expired, and killing shells if the user does not log out. A hangup signal is sent to all of the user's other processes, just as if he had logged out by turning the terminal off.

Login communicates with the MUSH daemon in order to ensure that the user about to login has sufficient login time available to be able to do something useful. If not, the login is refused.

5. Protection and Privileges

The UNIX filesystem provides adequate protection for a user who desires to protect his data from examination by other users. He may remove read permission from files or encrypt the data contained therein. However, there is no mechanism to prevent a user from granting permission to others to examine or copy his files, indeed, in most UNIX environments, such sharing is positively encouraged.

In our student environment however, where students are required to do their own work, and refrain from sharing what they have done with their friends, some control over this ability is necessary.

Our solution is quite simple. There is a user privilege (set by `login(1)`) which prevents a user from altering his file creation protection mask, and causes that to be applied to any attempt to change the protection bits on his files. Students are given an initial umask of 007,[†] and are thus prevented from ever giving access to 'others'.

Of course, by itself, this would pose only the most minor of hurdles to one dedicated to spreading his good works amongst his fellows. In this regard, we also have restrictions on who may receive mail from students and on use of the

[‡] Or a child of the remote login daemon.

[†] The group access bits are explained below.

various networking facilities in the system.

Then we need very tight security to ensure that there are no generally accessible files with read-write permission to everyone. This is eased by placing all students in the same group, and placing all other users in a segment of the filesystem tree that passes through a directory owned by the student group. Permissions on this directory are then set at 705 (no access for the group) and students are effectively barred. Thus, we need only check the, relatively static, public and system parts of the filesystem, and need not worry about careless users leaving files that students can write in.

Students' directories are made in a group that the tutors are a member of, but which the students are not. 4.2BSD creates files in the group that owns the directory that contains them. That, together with the initial umask of 007, allows tutors to access student files. Students may, of course, prevent access by their tutors, or permit only read access.

Students are prevented from changing the group ownership of any of their files, and there are no directories anywhere in the system that are writable, and in the student group, so there is no way that the open group access permissions can be used to share files.

This orthogonality of the permission bits, together with the excellent group scheme in 4.2BSD, enables this protection scheme to be implemented remarkably easily, and relatively robustly.

6. Administration

The systems described above are generally largely self administering. Student accounts are created by means of a special account making shell, used by the student herself. That program accesses a database provided by the university administrative section, and validates the student's right to access the system. Then it chooses a login name and user id, and makes her initial directory. Privileges are set by making a composite from the default privilege files maintained by the tutors for each unit offered.

Students may examine their privs, limits, etc with an enhanced version of the *quota*(1) command. *Edpriv*, a command similar to *edquota*(8) allows tutors to alter a students privileges as required, or the super-user to alter those of any user.

7. Performance

Kernel profiling indicates that the share scheduler uses approximately 0.04% of available cpu time. The MUSH daemon adds approximately the same overhead as /etc/init (process 1). That is considerably less than the pagedaemon (process 2), the swapper (process 0) or /etc/update.

One unfortunate consequence of the current implementation, is that because the IPC used transfers only very small data packets, quite a large number of messages are exchanged between *login* and the MUSH daemon. This slows logins, and adds some overheads.

The various data files are all indexed by user id, which can use vast amounts of disc space if uid's are allocated randomly. This hasn't bothered us, as we confine our uid's to a comparatively small subset of the available range.

8. Future Work

The MUSH system has been in operation, in various guises now for about 3 years. It is still not complete. Little is known about the effects of varying the cost parameters of the share scheduler, and this area warrants further study.

The share scheduler is well suited to the needs of an environment where there are many users, all making more or less continuous, and approximately, equal demands on cpu resources. However, it does not transport well to situations of less regularity, where users might make occasional large demands on the cpu, then remain idle for long periods. This area needs further work.

The MUSH daemon needs to be modified to use the UNIX domain IPC mechanism provided in 4.2BSD.

9. Conclusions

The MUSH system has proved itself to be most useful in assisting us survive with a very badly overloaded processor. It is certainly not a reasonable substitute for sufficient computing power, but it's the only substitute that we have.

The system is **not** currently available for distribution.

10. Acknowledgments

The work described above is all loosely based on similar work done at the Universities of New South Wales and Sydney in the late 70's. The share scheduler is almost a direct re-implementation of that work. The use of the MUSH daemon process, however, is new to this implementation.

Finally, I must thank the undergraduate students of Melbourne University, who have done their utmost to test every restriction or limit added to the system, and who have worked diligently to discover if there are any loopholes or errors in the implementation.

A Dynamic Bad-Block Forwarding Algorithm

Bakul Shah

458 Ferne
Palo Alto, CA 94306

Robert P. Warnock, III

Fortune Systems Corporation
Redwood City, California 94061

ABSTRACT

An abstraction of a more reliable disk can be provided by the combination of a logical-to-physical mapping function and the migration of good data from deteriorating disk blocks. The resulting system is much more robust than when mapping functions alone are used. An algorithm is given for the maintenance of the mapping information and for the data migration.

For every block on the disk we keep track of the state of logical-to-physical mapping, the state of the physical media for that block and the state of the data stored in it.

The algorithm is distributed between the kernel disk driver and a utility program. The disk driver keeps track of the state of each block, handles read-retries explicitly, allocates alternate physical blocks when a bad block is written, performs some (but not all) of the state transitions of the algorithm, and maintains a short-term log of any ECC corrections and state changes. The utility program "fixdisk" performs the remainder of the state transitions, maintains a long term error log, and performs the actual migration of data from any deteriorating blocks. The "fixdisk" program runs periodically (every day or so) and takes its policy either from table-driven parameters or from manual intervention.

Related issues discussed include procedures for initializing the bad block status on new drives, testing disk surfaces during initialization and during data migration, handling of errors in blocks used for "i-nodes", and management of the pool of blocks used for spares.

An Expandable Object-based UNIX Kernel

Erik Reeh Nielsen
Søren Lauesen
Vilhelm Rosenqvist

NCR Systems Engineering Copenhagen
Haslegade 16
DK-2100 Copenhagen
Denmark
phone: + 45 1 29 58 00
telex: 16450 ncrse dk

Abstract:

Many key features of UNIX are hidden in the UNIX-kernel. This implies that users cannot expand their system the way they need unless they modify the kernel.

A commercial UNIX* based system may require changes like the following:

- replace security policy
- add reliable database system
- add transparent remote file access
- replace or add file system to allow disc formats compatible with other operating systems.

If a source license is available, it may be feasible for one organization to make its own changes to the UNIX-kernel and to maintain the new kernel version. However, changes/extensions made by different organizations cannot easily be integrated under present kernel architectures.

This presentation shows a UNIX-kernel implementation, called the EOS Software Factory, that overcomes these problems. The UNIX-kernel is implemented as a number of separately protected modules (objects). The modules are dynamically coupled when the system executes. Modules may even be installed and removed on-the-fly.

Actually, the EOS software factory not only support UNIX, it can as well support applications written for other operating systems if the appropriate modules are added.

The EOS Software Factory has been implemented on standard hardware (Motorola 68000) and runs with competitive performance.

* UNIX is a trade-mark of Bell Laboratories.

Disclaimer

This paper explains an experimental operating system developed by NCR. It is not a product offering, but serves only as an introduction to the technology involved.

Introduction

EOS is the name of an object oriented software technology for decentralized operating systems. A decentralized operating system means an operating system where OS-modules like device drivers, file systems, database systems, and job handlers can be independently developed and easily combined. With the EOS technology OS-modules are separately protected, they can be user-written, they can be installed and tested on-the-fly like ordinary user programs, and they are dynamically linked together with no need for a special link or sysgen phase. To a system builder this means that he can easily tailor his system, e.g. get some OS-modules from the hardware vendor, other OS-modules from 3rd party, and add some OS-modules himself.

The EOS technology is initially available through an operating system build on these principles. This OS, called the EOS Software Factory, contains a decentralized implementation of the UNIX kernel. The Software Factory can be used to run the wealth of UNIX applications and development tools, it can be tailored for special purpose applications or support other operating systems than UNIX, and it can be used as development system for further OS-module development. The first version of the Software Factory runs on a Motorola 68000 Exormacs.

The acronym EOS stands for Expandable Operating System. In Greek the word EOS means "dawn" or the goddess of dawn.

State-of-the-Art Operating Systems

In a state-of-the-art operating system many key features, e.g. job handling, resource allocation, file management, and security management, are part of a single central operating system component usually called the kernel, executive, or supervisor. In UNIX this central component is the UNIX kernel.

Such features, that are part of the kernel, cannot be modified or extended without stopping the entire system, since changes requires recompilation or relinking of the kernel followed by a restart with the new kernel. Although the kernel internally consists of several modules, these are not protected, so an error in one module may easily harm the reliability of other modules. For these reasons changes to kernel features is a cumbersome and risky process, in particular in commercial production environments.

Features that are implemented as application (or utility) programs can, however, normally be added or modified in a much more smooth manner. Since applications normally execute in a protected environment, they can be tested without harming other parts of the system. In fact, a major function of most state-of-the-art operating systems is to allow users to develop and execute application (or utility) programs.

Most existing operating systems allow an application to communicate with other applications. This gives the application some means of providing services to other applications, i.e. act as an operating system component. UNIX has several examples of applications (utilities) serving as operating system components: The shell and printer spooler are two such examples.

The benefits from implementing operating system functions as utility programs on state-of-the-art operating systems are numerous. As explained above, OS-modules implemented as utility programs can easily be added or modified. Any user can be allowed to develop such an OS-module, since it executes in a protected environment and, thus, cannot harm other parts of the system. Several different versions of the same service can coexist, witness the various UNIX shell versions.

Unfortunately, the communication facilities between applications in state-of-the-art operating systems are insufficient or too specialized so that many OS-facilities cannot be put into utility-programs. In UNIX, for example, new OS-facilities like a communications driver, a reliable database system (with commit and roll-back), or a transparent remote file access facility cannot be implemented as utilities, they have to be integrated into the UNIX kernel.

Purpose of EOS technology

The overall purpose of the EOS technology is to make the central part of an operating system as small as possible. Most of the OS-modules become "decentral modules", that can be developed by the user or 3rd party. The remaining central part, called the EOS kernel, is the Software Bus which lets the decentral modules interface with each other.

More precisely, EOS fulfills the following requirements:

System software can be developed/added by user:

The user can combine 3rd party OS-modules, own OS-modules, and vendor modules, and he does not need the source texts of 3rd party or vendor modules.

System software modules cannot corrupt each other:

OS-modules can only interface with other modules through the EOS-kernel, and the kernel does not allow one module to change the interior part of another module. This property facilitates testing and isolates errors. Thus, it improves system reliability.

System software modules can be installed and configured on-the-fly:

An OS-module can be installed while the system is running normally. After installation, the OS-module can be instructed to configurate itself (adapt to the peripheral devices available, the expected load, etc.). An OS-module can also be removed on-the-fly; for instance, when in error or as preparation for a new version to be installed.

System software can be developed incrementally:

A new OS-module improving the services of an existing module can be installed and transparently made available to the users. The improved services can often be implemented on top of the old services without need for access to the source text of the existing module. A file system with memory cache, for example, can be built on top of an existing file system handling the disc without cache.

Several file systems, job handlers, etc. can coexist:

An EOS-based system can support several file systems; for instance handling different disc formats. Furthermore, an application can concurrently access some files through one

file system, other files through another file system. Also, several job handlers can run concurrently; for example one supporting UNIX and another supporting CP/M or MS-DOS.

EOS Technology Principles

The EOS technology is based on the object-oriented operating system technology pioneered by Hydra [Wulf 74, 81] and CAP [Wilkes 79]. A major achievement during the EOS development has been to improve this technology, in particular in the area of abortion and clean-up, and show how a decentralized, commercial OS could be organized.

The EOS technology can be characterized by the following points:

Object oriented:

An EOS module is an object or a collection of closely related objects.

An object is a collection of data and procedures operating on that data. An object can only be accessed from outside by means of the so called entry procedures. The data of an object can not be accessed directly from outside the object, but the entry procedures may return copies of the data or pointers to part of it. These rules are enforced by a combination of hardware protection and the EOS kernel or Software Bus.

"Large" objects:

EOS uses objects mainly for enforcing protection. Usually, an OS-module consists of a single or a few reasonably large objects. A file system is one object, an executing application program is one object, etc. This means that traditional programming style

can be used inside objects, and that the EOS kernel only intervenes when an object (protection) boundary is crossed during execution. The result is that acceptable performance is possible with traditional hardware.

Decentralized abortion and clean-up:

A major problem in decentralizing an operating system is abortion and clean-up. As an example consider a database system engaged in servicing an application. The database system has set aside resources for the application etc. When the application terminates, the database system must perform some clean-up (make resources free, roll transaction back, etc.). In case the application terminates in error or is aborted, it cannot itself notify the database system. Inventions made as part of the EOS development and implemented in the Software Bus can be used by any OS-module to ensure that it can clean-up when serviced modules terminate.

Another problem arises when an application is waiting for some resources within the database system (maybe waiting indefinitely because of a deadlock) and the operator wants to kill the application. The database system must then be notified so that it can cancel the waiting. The EOS development has also resulted in new inventions for solving this problem in a decentralized manner.

```

object IoSystem
  entry Assign (out file; name, .. )
  entry Rename (oldName, newName )
  entry CreateLink (oldName, .. )
  entry DeleteLink (fileName )

object File
  entry Read (buffer, pos, count )
  entry Write (buffer, pos, count )
  entry SetSize (fileSize, volume)
  entry SetMode ( .. )

```

Figure 1. EOS I/O System Family

Architecture for OS-module families:
 EOS shows how a complete, commercial operating system can be split into rather independent modules. In order to allow several file systems to be interchangeable, certain rules must be followed by a module qualifying as a (standard) file system. These rules define the family of file systems. How a file is organized on disc, etc. is not prescribed in the family rules. Only a set of interface procedures (entries) to the file system is part of the family standard. These interfaces are sufficient for an application program (or an OS-module) to use any file system belonging to the family. Figure 1 shows part of the I/O-System family specification.

Similar families are prescribed for resource managers, job handlers, security managers, etc.

Initial Implementation: EOS Software Factory

The EOS technology is initially available as the so-called Software Factory, an operating system built

on the EOS principles. The Software Factory contains a decentralized implementation of the UNIX kernel (in the first version compatible with UNIX version 7).

The EOS-UNIX-kernel is not built into the EOS-kernel. Rather it has just been implemented as OS-modules among other OS-modules. In fact, most of the modules comprising the decentralized UNIX-kernel are standard EOS modules useful for other purposes as well.

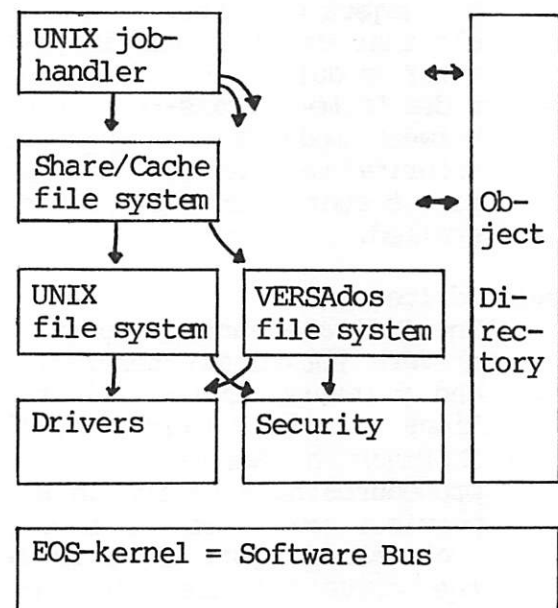


Figure 2. EOS Software Factory.

The EOS Software Factory consists of the following software modules (see figure 2):

The EOS-kernel:

This Software Bus provides the foundation which lets the decentral modules interface to each other, it enforces protection, and it includes basic memory management and process

scheduling. The basic memory management and process scheduling are used by decentral OS-modules which provide more elaborate resource management and scheduling.

Security module:

This module implements a security database for use by the other OS-modules. It is optional in the sense that meaningful defaults are assumed by the OS-modules in case the security module is not installed.

Object directory:

The object directory is a module that can load and install other modules. It also provides named cross-references between modules - subject to security restrictions in case the security module is installed.

Device drivers:

The Software Factory contains drivers for discs, terminals, and printers. Due to limitations in the Motorola 68000 EXORmacs hardware, interrupt procedures have to run in supervisor mode without memory protection. Apart from this, the driver modules are not different from other OS-modules. They can be installed and removed on-the-fly and they are written in a high-level language (EosPascal).

UNIX and VERSAdos file systems:

The software factory contains two standard low-level file systems, one supporting UNIX disc packs and another supporting Motorola VERSAdos disc packs. Both file systems can be used via normal UNIX i/o procedures.

File sharing and cache module:

UNIX-type file sharing and disc cache is implemented in a separate module, which can use any standard i/o-system for low-level file i/o. This module, the device drivers, and the file systems all belong to the i/o system family. As a result an application program (or OS-module) can use these modules interchangeably. For instance, it can bypass the cache by using one of the low-level file systems directly. UNIX system calls concerned with i/o is mapped into standard EOS i/o-system entries by the system library procedures.

EOS-UNIX job control module:

This is the only special purpose module in the decentralized EOS-UNIX-kernel. It handles UNIX job and process management, i.e. login, fork, execute, signals, etc. It relies entirely on standard mechanisms in the EOS-kernel. UNIX system calls not concerned with i/o is mapped into entries in this module.

The Software Factory modules can be installed and removed on-the-fly. For instance, when demonstrating EOS, we usually show the following steps:

- Boot a system consisting of the EOS-kernel, the Object Directory, terminal and disc drivers, the VERSAdos file system, and a simple native EOS job handler (not part of the software factory).
- Using the native EOS job handler we execute a price-look-up application handling two terminals. We then install a cache file system on-the-fly and let one terminal use it.

The resulting performance improvement is striking. We also show how the new module responds properly when the application is aborted.

- After this we install the rest of the Software Factory modules and dynamically configure the EOS-UNIX job control module to use terminals not used by the native EOS job handler.
- The EOS-UNIX-kernel now runs concurrently with the native EOS job handler. In fact, applications running under either job handler can use much the same facilities (they can use the same decentral OS-modules). UNIX facilities like fork, exec, signals, etc. are, however, only available to applications running under the EOS-UNIX-kernel. Next, the native EOS job handler is removed to make all system resources available to the EOS-UNIX-kernel.
- After installation of a Window Terminal System (not part of the Software Factory), still without stopping the system, we show that it is possible to run UNIX from multiple windows on each terminal.

The first implementation of the EOS Software Factory has been made on a Motorola 68000 EXORmacs. It can relatively easily be moved to other hardware.

The EOS-kernel is written in assembly language and occupies about 30K bytes of storage. The OS-modules are written in EosPascal, an extended version of Pascal, and occupy about 150K bytes. The majority of UNIX applications and utilities are of course in "C", since we use the AT&T text unchanged. Each UNIX job occu-

pies 1.5K bytes of system data, and each UNIX process 1K of system data.

The performance seems acceptable: On our first prototype the CPU-overhead to perform a disc file i/o is about 10 - 30 milliseconds (depending on the file systems used) on an 8 MHz MC68000. We expect these numbers to be halved when implementing planned improvements to the EOS-kernel.

Conclusion

We have introduced a new operating system technology that makes it possible to structure an operating system into decentral, protected modules. This technology has been used to implement an operating system, the EOS Software Factory, including a decentralized UNIX kernel.

While maintaining UNIX compatibility for existing tools and applications, the EOS Software Factory represents a significant improvement in flexibility.

Acknowledgements

The design and technology behind EOS was initiated at the University of Copenhagen by Eske B. Andersen, Ole Caprani, Søren Lauesen, Anders Peter Ravn, and Erik Reeh Nielsen.

The design presented in this paper was made at NCR by the authors and Peter Mikkelsen. The present implementation was made by the same four persons together with Charlotte Lunau and Edith Rosenberg.

References

[EOS 83]

S. Lauesen, E. Reeh Nielsen, V. Rosenqvist: EOS-Kernel Reference Manual.
NCR SE-Copenhagen Document No. 905-0000091.

[EOS 84]

EOS: System Programmers Introduction. NCR SE-Copenhagen Document No. 905-0000184.

[Wilkes 79]

M.V. Wilkes, R.M. Needham: The Cambridge CAP computer and its operating system. Elsevier North Holland, 1979.

[Wulf 74]

W. Wulf et.al.: Hydra: The kernel of a multiprocessor operating system. Comm. ACM 17,6 (June 1974) pp 337-345.

[Wulf 81]

W. Wulf et.al.: Hydra: C.mmp: An experimental computer system. McGraw-Hill, 1981.

Appendix: EOS kernel mechanisms.

The most important EOS kernel mechanisms for module interaction will be illustrated by selected parts of an example OS-module that implements file reservation.

Object:

The interface of the file reservation module adheres to the standard i/o-system family architecture (cf. figure 1). Each open file is represented as an **object**. The file reservation module thus consists of a main i/o-system object and a variable number of file objects. Figure 3 shows the specification for the file objects. It includes a specification of the protected data of the objects (lowLevelFile and localData) and the code of the entry procedures (Read etc.).

The file reservation module uses another standard i/o-system for file storage. The file object used to store the contents of the file is referred to by the **kernel-pointer** (capability) lowLevelFile.

```

program ReservedFile object File
  with record ...
    lowLevelFile: ^^File;
    localData: ...
  end;
  entry Read {buffer, pos, count}
    ...
  begin
    ...
    lowLevelFile.Read
      (buffer, pos, count);
    ...
  end { Read } ;
  ...
end { ReservedFile } ;

```

Figure 3. File object implementation

Object Calls:

File i/o is done by calling the entry procedures of a file object (see figure 4). For UNIX programs these calls are part of the system call library procedures.

```

filel.Read (buf, readpos, count);
filel.Write(buf[l..count], writepos,
            ignorecount);

```

Figure 4. File object entry calls.

An entry procedure is called using the **object call** kernel operation. An object call appears much like an ordinary procedure call, but in fact it involves a change of hardware protection environment. When a process executes the code in figure 4 it cannot access the protected data of filel, it can only call the entry procedures. After calling filel.Read the process executes within the filel object. It can access the protected data and in turn call lowLevelFile to perform the actual i/o (cf. figure 3).

Object Creation:

To open a file the Assign entry in the i/o-system object of the file reservation module has to be called. The Assign entry creates a file object to represent the open file. Figure 5 shows the implementation of the Assign entry of the reservation module.

Multiple users may have read access to a file but to write to a file a user must have exclusive access. The "reserve" procedure lets the calling process wait until the required "iorights" can be granted (ignoring for this example the risk of deadlock).

```

entry Assign { file, name, iorights}
...
begin
...
  Reserve (name, iorights);
...
  alloc.NewObj (file, ...);
  {Call allocator object to
    allocate space for the new
    file object}
...
  DeclGen (file, filedata, ...
    ReservedFile, ...
    Close, ... );
  {Declare new object to be a
    file object. "ReservedFile"
    specifies implementation, and
    "Close" is termination
    procedure}
...
  lowLevelFileSystem.Assign
    (filedata^.lowLevelFile, ...)
  {Assign low-level file storing
    contents of file}
end { Assign } ;

```

Figure 5. Assign entry.

The space for the new object is allocated by an allocator object. This

may be the primitive allocator which is part of the EOS-kernel or it may be a more sophisticated module implemented on top of the primitive one.

The kernel operation **DeclGen** is used to declare the new object to be a file object. It defines the protected data and entry procedures of the object by specifying "ReservedFile" (cf. figure 3) to be the implementation of the object. It also specifies "Close" to be the termination procedure to be called when the object is deleted (see below).

As part of the initialization of the new file object the low-level file storing the contents is assigned.

Object Deletion:

An open file is closed by deleting the corresponding file object. This may be done explicitly (using the **Dealloc** kernel operation) or implicitly (as a result of the object assigning the file, e.g. a job, being deleted).

When an object is deleted the termination procedure specified during creation is automatically called. Figure 6 shows the termination procedure "Close" specified in figure 5.

```

private Close (file, filedata)
...
begin
...
  Release (filedata^.localdata);
  {Cancels reservation made by
    "reserve"}
...
end { Close } ;

```

Figure 6. Termination procedure.

Once a file object has been created the EOS-kernel ensures that its termination procedure is called when it is deleted. Since a file object is deleted no later than the object creating it, i.e. the object opening the file, the reservation module gets the opportunity to clean up and cancel the reservation regardless of whether the user remembers to close the file. Note that "Close" does not explicitly delete "lowLevelFile", it is automatically deleted during deletion of the "ReservedFile" object.

Abortion:

When a module is **aborted** the entry procedures of the objects defined by the module is made empty, i.e. processes executing or calling the objects will return immediately.

As an example consider an eternal loop in the "Close" procedure of the reservation module. This will make a process trying to close (delete) a file loop until the reservation module is aborted. After the module has been aborted "Close" returns immediately. The deletion of the file object then automatically proceeds with the deletion of the low-level file object (including a call of the termination procedure in the low-level file system).

Processes that have called an aborted module does not necessarily return immediately, since they may have to perform some clean-up in non-aborted modules (as can be seen from the above example). However, the processes will return eventually, provided that these modules are ok.

Speed-Up:

An object called from an aborted object may not have to complete its operation in the ordinary way.

As an example consider the "Reserve" procedure of the file reservation module (figure 7). If the object opening a file is aborted there is no need to wait for the ioRights to become free, since the file will never be used.

When an object is aborted a **speed-up signal** is propagated into objects called by the aborted object. The speed-up signal can be tested explicitly or be allowed to continue to propagate so that later operations may be speeded-up, as in figure 7.

The "Reserve" procedure takes advantage of the fact that the "Wait" entry cancels its waiting and returns with a special result when receiving a speed-up signal. Note that even after being speeded-up the procedure has to do some clean-up, i.e. adjust "waitCount".

```

procedure Reserve ( name, ioRights )
    ...
begin
    ...
    if rightsOccupied then begin
        waitCount := waitCount + 1;
        propagationMode (reject);
        {Allow speed-up signal to
         propagate so that "Wait"
         may be speeded-up}
        res := rightsFree.Wait;
        {May be speeded-up}
        propagationMode (stop);
        {Prevent speed-up signal
         from propagating}
        waitCount := waitCount - 1;
        if res = speededUp then
            Return; {Without creating
                     file object}
    end;
    ...
end { Reserve } ;
  
```

Figure 7. Reserve procedure.

Processes as Files

T. J. Killian

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We describe a new file system, */proc*, each member of which, */proc/nnnnn*, corresponds to the address space of the running process whose pid is *nnnnn*. Access to these files is restricted, via the normal file protection mechanism, to the process owner. *Lseek(2)*, *read(2)*, and *write(2)*, allow inspection and modification of the process' image. Other services are available via *ioctl(2)*, including stop/go on demand, selective intercepting of signals, and the ability to obtain an open file descriptor for the process' text file. The technical problems related to the implementation of */proc* on a VAX[†] under the 8th Edition of the Unix[‡] operating system have mostly to do with the paging system. Security issues are also considered.

The window-based interactive debugger *pi*, developed by T. A. Cargill, is the first major user of */proc*. It can control multiple processes dynamically and asynchronously. We describe it briefly, and discuss its system interface.

Introduction

Any debugger is dependent on, and often limited by, its ability to access the address space of the debugged program. This is especially true in the case of interactive debugging under the Unix system, where the debugger and the debugged object are separate processes. The problems associated with the standard mechanism, *ptrace(2)*, are well known:

- The object must agree explicitly to be debugged, and furthermore it can only be debugged by its immediate parent. Thus there is no dynamic binding, and children of the original object process cannot be handled.
- Before it can be examined, the object must be put in a stopped state, typically by sending it a *signal(2)*. This can interrupt the object's own system calls, so the debugging is not transparent. Or the object may be ignoring signals (e.g., sleeping forever on a locked inode), so the mechanism can fail entirely.
- *Ptrace(2)* provides low bandwidth at high cost: two context switches per word of data transferred, an achievement equaled only by some text editors. Its protocol is arcane and unnatural compared to most other system calls.

We have tried to overcome these difficulties by providing an interface that is as uniform as possible, using an existing mechanism for accessing random data external to a process: the file system.

[†] VAX is a trademark of Digital Equipment Corporation.

[‡] Unix is a trademark of AT&T Bell Laboratories.

Fig. 1: A sample `/proc` directory

```

-rw----- 1 root      14336 Feb 20 12:59 00001
-rw----- 1 root      528384 Feb 20 12:59 00002
-rw----- 1 root      12288 Feb 20 12:59 00019
      :
-rw----- 1 tom       32768 Feb 20 12:59 02596
-rw----- 1 tac       106496 Feb 20 12:59 02652
-rw----- 1 root      14336 Feb 20 12:59 02801
-rw----- 1 tac       39936 Feb 20 12:59 02900
-rw----- 1 tac       23552 Feb 20 12:59 02910
-rw----- 1 tac       184320 Feb 20 12:59 02911
-rw----- 1 tom       33792 Feb 20 12:59 02912
-rw----- 1 tom       54272 Feb 20 12:59 02913

```

System-Call Interface

Fig. 1 shows the result of a typical `"ls -l /proc."` The name of each entry in the directory is a five-digit decimal number corresponding to the process id. The owner of the "file" is the same as the process' user-id; note that only the owner is granted permissions. The size is the total virtual memory size of the process. The time is not very useful: it is always the current time, for reasons discussed later.

The standard system-call interface is used to access `/proc`. `Open(2)` and `close(2)` behave as usual with no side-effects. In particular, the object process is not aware that it has been opened. Data may be transferred from or to any locations in the object's address space through `lseek(2)`, `read(2)`, and `write(2)`. Reading and writing have slightly peculiar behavior due to the segmenting of the process' address space.

The *text segment* (see Fig. 2) will be read-only, if it was already shared at the time of the attempted write; otherwise, the text is marked impure, and subsequent writes are guaranteed to succeed. The *data* and *stack segments* are read/write in any case. The *user area* is read-only, except for locations corresponding to saved user registers. The system segment is not accessible. For simplicity in enforcing these restrictions, a single I/O operation may not cross a segment boundary; the byte count will be truncated if necessary. Note that for ordinary files, the entire file is either write-protected or not, and "holes" read as zeroes.

As with other special files, there are a number of services available via `ioctl(2)`, in this case having to do with process control:

PIOCGETPR fetches the object's `struct proc` from the kernel process table. Since this information resides in system space, it is not accessible via a normal read.

PIOCSTOP sends the signal `SIGSTOP` to the object, and waits for it to enter the stopped state.

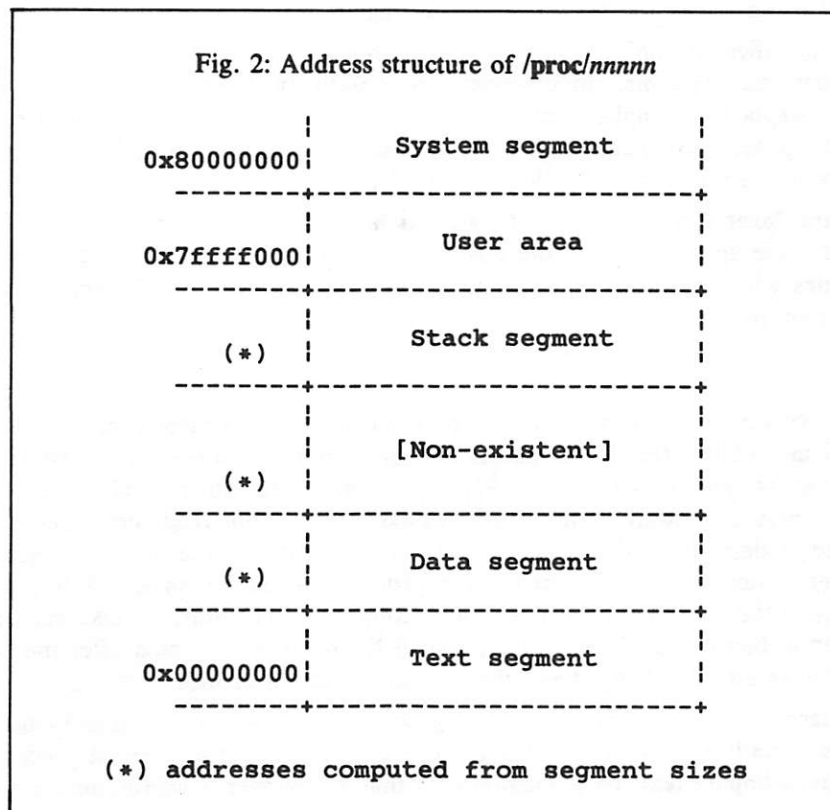
PIOCWSTOP simply waits for the object to stop.

PIOCRUN makes the object runnable again after a stop.

PIOCSMASK specifies (via a bit mask) a set of signals to be "traced"; i.e., the arrival of such a signal will cause the object to stop. A mask of zeroes turns off the trace. There are two side-effects: (1) a "traced" process will always stop after exec'ing; and (2) the "traced" state is retained after the object is closed, although the mask bits themselves are lost.

PIOCCSIG clears all of the object's pending signals.

PIOCEXCLU marks the object's text segment as impure, so that subsequent writes will succeed. This `ioctl` fails if the text is already shared.



`PIOCOPENT` provides, in the return value of the `ioctl`, a read-only file descriptor for the object process' text file. This allows a debugger to find the symbol table without having to know any path names.

All system calls are interruptible by signals, so that, for example, an `alarm(2)` may be set to avoid waiting forever for a process that may never stop. Any system call is guaranteed to be atomic with respect to the object, but, as with ordinary files, there is nothing to prevent more than one process from trying to control the same object.

Implementation

The interface to the file system was provided by P. J. Weinberger, who introduced the notion of a *file system type* to support his network file system [1]. In a sense, this is a generalization of the top level of the `pipe(2)` mechanism, in which, e.g., the kernel's read routine calls a special procedure if the file descriptor refers to a pipe. In Weinberger's scheme, the kernel has a well-defined set of internal entry points (*read*, *write*, *open*, *close*, etc.) for each file system type, and uses the appropriate one transparently. A special `mount(2)` command is used to associate a particular file system type with a given leaf of the directory hierarchy. Weinberger in fact made an early attempt at an implementation of `/proc`, but the communication mechanism was too similar to that used by `ptrace`, making it impractical.

In our implementation, the calling process accesses the object directly. This requires the cooperation of the swapper, the scheduler, and the paging system. A flag in the object's `struct proc` informs the system globally that the object is undergoing I/O via `/proc`. The swapper recognizes the object as a candidate for being swapped in, and will not swap it out as long as the flag is set. The scheduler will not run the object; in particular, the effect of any signal or wakeup on the object will be delayed until I/O is completed. Finally, hooks have been added to the paging system so that the object's pages may be brought in on demand by the calling process. Thus I/O via `/proc` can take place under almost any circumstances. The exceptions occur where the object is waiting for an actual virtual-memory event, i.e., it is exec'ing, paging, forking, or exiting. In these cases,

the I/O call returns an error, and the caller must try again.

Because the caller and object must both be swapped in during I/O, there is the possibility of deadlock on very small systems. In practice, this is unlikely to happen under 8th Edition Unix because being swapped in implies only that the user area and page tables are present; the remainder of the process can page in and out as necessary. In any case, I/O via `/proc` is always interruptible, even in an otherwise deadlocked situation.

Reading the `/proc` directory and `stat`'ing(2) its members present special cases. The information returned is made up exclusively from the kernel's process table. Some potentially useful data, like process times which reside in the user area, are not returned for efficiency reasons, since a swap might be required to get access.

Security

The most obvious security loopholes are plugged by the file system itself. The standard protection mechanism prohibits free-for-all access to every process, and the file system type itself does not support things like `mv(1)`, `rm(1)`, `chmod(1)`, etc. Some more subtle problems are the following: `/proc` provides a way of reading a file which has execute, but not read, permission; this is easily solved, since the inode of the object process' text file is available at the time the object is `stat`'ed or opened, and permissions can be checked. If a process opened by `/proc` exec's a program with `setuid` bits, there is the potential for any user to become the super-user; we take care of this by not honoring the `setuid` bits in such a case. Note that if the open is attempted after the exec, the process owner will have already changed and the normal protection applies.

Finally there is the problem of corrupting shared text, which has already been discussed. This problem is actually solved over-zealously, as a convenience: if an opened process exec's, it is automatically given impure text, on the assumption that a debugger is waiting in the wings.

The *pi* system interface

T. A. Cargill has developed an interactive, window-based debugger called *pi* (process inspector). It can be bound dynamically to multiple processes, and control them asynchronously. Binding is particularly simple: *pi* opens the object process, gets hold of the text file via `PIOCOPENT`, and reads the symbol table. It is frequently useful to have control of the object before it has actually begun executing. This is easily done, given the semantics of `PIOCSMASK`. Fig. 3 shows the program *hang*, which takes any command as argument, and starts it up in the stopped state.

Pi obtains status information and data from a process using `ioctl`'s, or a combination of seeks and reads. Other operations require a more complicated series of primitives. *Single-stepping* is accomplished as follows (the object is assumed to be already stopped):

- 1) Arrange, via `PIOCSMASK`, for the object to stop on `SIGTRAP`.
- 2) Set the `TRACE` bit in the object's `PSL`. This involves a seek, read, and write in the object's user area.
- 3) Issue `PIOCRUN`.
- 4) Wait for the object to return to the stop state, by issuing `PIOCWSTOP`.
- 5) Issue `PIOCSIG` to clear the `SIGTRAP`. (If the object has other signals pending which should be preserved, they can be read via `PIOCGETPR`, and resent.)

Single-stepping over a function call (`CALLS` instruction) is only slightly harder. If the instruction stepped was a call, one sets the `TRACE` bit in the `PSL` saved on the stack and issues `PIOCRUN`; the `SIGTRAP` will then be taken upon return.

Breakpointing is similar to single-stepping. In step (2) above, one writes a `BPT` instruction at the desired location.

Fig. 3: The *kang* program

```

#include <stdio.h>
#include <signal.h>
#include <sys/pioctl.h>

main(argc, argv)
int argc; char **argv;
{
    int pfd; char procnam[16]; long mask = (1<<(SIGSTOP-1));
    FILE *ttyerr;
    ttyerr = fopen("/dev/tty", "w");
    if (argc <= 1) {
        fprintf(ttyerr, "Usage: %s cmd [args...]\n", *argv);
        exit(1);
    }
    sprintf(procnam, "/proc/%05d", getpid());
    if ((pfd = open(procnam, 0)) < 0) {
        fprintf(ttyerr, "cannot open %s\n", procnam);
        exit(1);
    }
    ioctl(pfd, PIOCSMASK, &mask);
    close(pfd);
    fprintf(ttyerr, "%s\n", procnam);
    fclose(ttyerr);
    execvp(argv[1], argv+1);
    perror(argv[1]);
    exit(1);
}

```

Summary

We have described a uniform mechanism for transparent, dynamic communication between a debugger and its debuggee, along with a set of primitives for process control. It is successful largely because it fits so well into the Unix model. We feel that it is probably ill-suited for general inter-process communication, though it should work well for specialized application-oriented debuggers, or other programs which would benefit from clean access to dirty data structures. For example, we have written a version of *ps(1)* which is about four times faster than the standard one.

Acknowledgements

It is a great pleasure to acknowledge many useful and stimulating discussions with T. A. Cargill, D. M. Ritchie, and P. J. Weinberger.

References

- [1] Weinberger, P. J. "The Version Eight File System," in *Proceedings of the Summer 1984 USENIX Conference*, Salt Lake City, Utah.

MEMORY MANAGEMENT UNITS and the UNIX* KERNEL

*Clara S. Lai
Chris Peer Johnson*

UniSoft Systems
739 Allston Way
Berkeley, CA 94710

INTRODUCTION

UNIX system performance depends on many factors, one of which is the memory management unit (MMU). Porting the UNIX kernel to MC68000/MC68010 based systems has given UniSoft a unique insight into the implementation and performance of MMU designs. This paper describes the kernel's interaction with the MMU, observations and recommendations of MMU design and implementation, and a brief discussion of the memory-management schemes UniSoft has come across.

The following terms are used in this paper. In the simplest case, a "process" is a program, "multiple processes" are concurrent programs, and "switching processes" is the transition from executing one program to executing another. A "memory management unit" is the hardware providing memory management to the system. A "context" is the hardware concept corresponding to a "process", e.g., having an MMU with 8 contexts means 8 processes can be "mapped" or "made known" to the memory management hardware at a time.

KERNEL PERFORMANCE and the MMU

Before discussing how the MMU affects kernel performance, it should be pointed out that the UNIX kernel performance depends on a wide range of factors. The MMU is only one of several factors necessary for a high-performance UNIX system. Several hardware components affect system performance; these include the CPU instruction cycle speed (8, 10 or 12.5 MHz), disks and disk controllers[†], serial controllers[‡], data transfer-rate and bus features[§], as well as the number of CPU wait states introduced by the MMU per memory cycle. Other factors affecting performance less directly include the amount of memory available (which, in turn, affects the amount of swapping), the amount of decision-making required in the interrupt service routine^κ, file system parameters and disk organizations, and the software required to implement the MMU. All of these factors affect UNIX kernel performance, to different degrees. This paper focuses on the impact of the MMU design.

The MMU in a UNIX system provides several basic services to the kernel. If these services were implemented entirely in software, they would require tremendous overhead. An MMU makes physical memory easily accessible to many processes simultaneously with minimal

* UNIX is a trademark of AT&T Bell Laboratories.

† e.g., a controller with DMA performs better.

‡ e.g., terminal controllers that interrupt processors for each character slow the system.

§ e.g., whether the speed of the bus can keep up with the processor and memory.

κ e.g., more logic is required to implement terminal controllers with a single interrupt vector for receiver and transmitter than for those with separate interrupt vectors.

system overhead. Each time a process references a memory address, the MMU translates this logical address into a physical address. In addition to protecting user programs from each other, the MMU also protects the UNIX kernel itself from faulty user programs. With the help of a MMU, each process runs without knowledge of or hindrance from other processes, and, at the same time, possibly sharing memory with cooperating processes. The more completely these services are implemented in the hardware, the faster the system performs.

Memory management hardware is generally designed for speed. Most of the proprietary MMUs are well designed, resulting in fast (zero-wait-state) memory access. However, any overhead generated in programming the MMU slows system performance. Different MMU designs require vastly different amounts of management software in the kernel. The less work the kernel has to do, the better the system performs.

The UNIX kernel performs operations requiring high-frequency interactions with the memory management unit. These operations happen many times a second, practically every time the user does data read/write, at every I/O interrupt, and at every clock interrupt. The following list summarizes the areas where the UNIX kernel and the MMU interact. These areas include memory allocation and mapping, switching processes, and accessing user memory (which can be subdivided into validating a user logical address and translating it into a physical address, and reading/writing the user memory). The kernel's objective is to perform these functions as easily as possible.

- *memory allocation and mapping*

Every time a new process is created, the kernel allocates chunks of physical memory for the process, locates the registers where the pieces of memory are to be mapped, then maps in the physical addresses as user virtual addresses. Ideally, the MMU should not place any requirement on memory allocation, the MMU registers should be easy to locate, and the contents that need to be loaded into the registers should be evaluated simply. Conversely, the following require additional logic in the kernel to decide where to map memory: size and boundary mapping restrictions, collision of MMU locations (i.e., more than one virtual address for one or more processes mapped in a given MMU location); multiple locations where a given virtual address can be mapped; and complicated formulas for locating and evaluating MMU registers or locations. Other pitfalls for MMU designs in this area include requiring a huge amount of memory for in-memory page tables, overly restricting kernel virtual address space, or having portions of the MMU registers dedicated for a particular purpose.

- *switching process*

Switching from the execution of one process to another involves setting up the MMU hardware to map in the new process and then switching to it. This is typically done by loading several special MMU registers. The number of processes that can be simultaneously set up in the MMU hardware varies from one to many. The number of registers that need to be set up per process also varies. If only one process can be mapped at a time, the mapping information for the new process has to be set up every time the kernel switches processes. If the MMU can map in two processes at the same time, then the kernel does not need to set up mapping information every time it switches between these two processes.

The objective is to reset registers infrequently and easily. Allowing many process to be mapped at the same time, keeping down the number of registers that need to be reset for each process, and avoiding implementing part of the context-switching mechanism in software helps keep this operation simple. Any added complexity, such as disabling supervisor mode when switching context, specifically needing to unset previously mapped areas before re-use, clearing status, waiting for acknowledgement and response, checking for return value and errors, etc., increases overhead and should be done in hardware, if possible.

- *validating a user logical address and translating it into a physical address*

The kernel constantly needs to read/write bytes, words, or blocks from and to user memory. Before actually doing the read/write, the kernel checks that the logical address referenced is within the user virtual address space. Validation is usually followed by translation into the physical address used by the kernel to access the data. For most MMUs, validation and translation simply require looking up one or two registers, or using built-in functions (as in the Motorola 68451). Requiring complex logic to find MMU location in a multiple-location scheme, and/or to interpret register contents once they are found adds to the overhead. For systems in which the kernel cannot read the MMU registers (write-only), the kernel either needs additional logic to keep track of the mapping information, or might need to actually map in the area through a scratch page and try touching it for validation.

- *reading/writing user area*

The amount of work the kernel has to do here can be viewed in three levels of difficulty. In the simplest situation, the kernel is a logical extension of the user (i.e., the kernel and the user are in the same virtual space). Moving information back and forth between the user and kernel is as easy as moving information within the kernel itself and does not even require translating into physical address. An example of this is the Stanford MMU. In the second level of difficulty, the MMUs allow one-to-one mapping of virtual to physical memory for the kernel. The user virtual address is translated into a physical address; once this is done, the physical address, which is the same as virtual for the kernel, is used to access the area as if it is part of the kernel area. An example of this is the Motorola 68451. In the third case, the MMU does not allow mapping in all of physical memory for the kernel. The user virtual address is translated into physical address, then this physical address is mapped into the kernel virtual space before actually reading or writing it. In this last case, mapping in the user physical as kernel virtual requires additional checking to determine if information is crossing page boundaries.

TYPES of MMU

The most popular MMUs UniSoft has implemented software for are the Stanford MMU and the Motorola 68451 MMU. Among the proprietary MMUs, the most commonly found are derivatives of the Stanford MMU. These derivatives range from those running the kernel out of a particular context to those eliminating segment tables. In this discussion, the MMUs are grouped as follows:

- Stanford MMU
- Motorola 68451 MMU
- Modified Stanford MMU and Single-level Stanford-like MMU
- Other proprietary MMUs
 - e.g., base and bound,
 - multiple base and bound,
 - a slave processor handling MMU
- no MMU^A

The amount of work it takes to implement the MMUs listed above is briefly described here, to provide a sense for the magnitude of possible diversity from a generic kernel. The modifications to the kernel required by the different MMUs can be viewed in three categories. In the worst case, there is no memory management unit. Since UNIX requires that processes

^A The requirements for this type of system are very different from one with an MMU; details are not discussed in this paper.

be switched, massive modifications throughout the kernel are required for it to run with no MMU. This includes major surgery, such as discarding the swap scheduler (process 0) itself. In the best case, implementing the MMU requires changes only in the MMU-dependent section of the kernel. This is fairly straight-forward and does not require changes outside of the MMU-related code. An example of this is the Stanford MMU. Between these two categories are MMUs that require modifications of code in parts of the kernel that are not directly MMU-related, such as memory allocations. An example of this is the Motorola 68451 MMU, which requires a particular memory allocation scheme. The following is a brief discussion of the various groups of MMUs; the intention is not to describe any particular MMU in detail, but to highlight some of the characteristics related to the kernel-MMU interactions noted above.

I. *The Stanford MMU*

The Stanford MMU has a multi-level mapping scheme. The highest level identifies the process number, called the context. For each context, the entire virtual address space of 2 megabytes is divided into many 2k-byte pages. These pages are accessed through segments (the next level), which point to page entries (the bottom level) in a page table. Each running process has its own page tables, whose entries point to fixed-size pages. Every time a memory address is referenced, it is translated to a unique page table entry containing the physical address. For most current implementations, translation from virtual to physical address in this mapping scheme is performed so fast that the 68000 processor requires no "wait states".

One characteristic of the Stanford MMU which does not occur in most other MMUs is the presence of the UNIX kernel in the user process' virtual address space. The supervisor does not run in a special context, but is mapped into virtual addresses in the same context as the current running process, using one set of page registers which is pointed to by each context through its segment registers. Using part of the user virtual address space for the kernel slightly reduces the addressable space for a particular process, but does not pose a real problem since the reduced virtual space is still quite large^a. A disadvantage of this feature is that binaries from a system with a Stanford MMU cannot be ported to systems with most other MMUs. In most other systems, the kernel does not take up any space in the user process' virtual address space, and user programs are usually originated at virtual zero. Since the kernel on a Stanford MMU system resides in low virtual memory of a user process, the process can no longer be originated at virtual zero.

II. *The Motorola 68451 MMU*

Unlike most MMUs which have unique direct correlation between virtual address and MMU register address, the Motorola 68451 has 32 descriptors, each of which can be used for any virtual address. Thus, a particular virtual address can be mapped using any one of the 32 descriptors; most other MMUs have a unique register where the virtual to physical information is kept. The 68451 has variable page size. Once the page size is selected, however, it restricts memory size and boundaries allocation; e.g., if a system chooses a page size of 2k, every mapping has to be of a size divisible by 2k. In addition, the boundary at which this piece of memory lies must be on a multiple of the size^b. In general, a system with one Motorola 68451 is quite well suited for four to eight users, with approximately 5 segments per process.

As mentioned before, on a system with Stanford MMU, virtual to physical address can be evaluated easily because the virtual address maps into unique segment and page register. On the Motorola 68451, however, a virtual address may be mapped into any of the descriptors, so

^a This reduces the addressable space from two megabytes to approximately one and a half megabytes.

In UniPlus[®], the kernel resides in location 0-80000 for a system with a standard Stanford MMU on 68000.

^b This creates certain problems. If one wants to map in a large process, this limits where it can be placed (e.g., if you want to map in a chunk of memory of 64k, this piece must go on a 64k boundary).

more searching is required. The result is comparatively slower address translation for the Motorola 68451 than the Stanford MMU. A maximum of 32 different chunks of memory can be mapped at a time. Due to the restriction on size and boundary, the standard memory allocation scheme cannot be used, resulting in a slightly higher overhead in the kernel for memory allocation^c. Memory also fragments more often under this mapping scheme.

These shortcomings are compensated for by several positive features. One advantage is fewer registers need to be set up. Special built-in functions also help to speed up several operations performed frequently by the kernel, including validating virtual addresses and translating virtual addresses into physical addresses. Despite these advantages and disadvantages, the Motorola 68451 and the Stanford MMU run similarly as far as MMU operations are concerned.

The above analysis is based on our experience with a swapping kernel; when implemented as a virtual kernel, there is a further disadvantage for the 68451. For a system with virtual memory, it is desirable to map in a lot of memory in reasonably small chunks at a time. Due to the small number of descriptors in a 68451, only a few chunks of memory can be mapped at a time. If these chunks are large, more memory can be mapped at a time, but the resolution on the page-frames is too rough for an efficient paging scheme. If a small page size is chosen, only a little memory can be mapped in at a time, and the system will page fault frequently. Compared to the 68451, the Stanford MMU scheme—with its fixed page size in small granularity—is better suited to a virtual memory system.

III. *Modified Stanford MMU and Single-level Stanford-like MMU*

A large group of proprietary MMUs fall into this category of variations of the Stanford architecture. A modified Stanford MMU has the same multi-level mapping as the Stanford architecture, except the supervisor runs in a separate context. A Stanford-like MMU does not have segments, but has a single-level mapping scheme with only page tables. In addition to these changes, other modifications are used in the proprietary MMUs for such things as to increase the number of processes mapped in at a time, to increase flexibility in accessing user memory, to provide faster translation of virtual to physical memory, to increase protection flexibility, and to increase addressable virtual space for a process.

There are a variety of approaches to achieving additional flexibility. These include allowing more contexts, allowing larger page sizes, allowing more segments per context, using in-memory page tables, using cache memory, using a common page table for all processes, allowing variations of mapping modes, allowing fancier protection permissions, and allowing the kernel to readily access the area of any context at any time. Most of these efforts do serve their purpose, the one exception being increase in protection flexibility. Since the UNIX kernel currently only makes use of write protection for shared text and shared data, additional protection permissions are not currently used.

Some of these modifications, however, introduce new overhead in the kernel. Examples are additional logic in the kernel to resolve collisions in a common page table, to invalidate pages in cache memory when switching processes, to handle insufficient page table entries, to do additional book-keeping for switching context, and to evaluate added levels of indirection for accessing MMU registers.

IV. *Other proprietary MMUs*

In addition to the MMUs listed above, we have encountered other schemes. Some of these are the more conventional base and bound or multiple base and bound schemes. Others use one or a few sets of registers to do the mapping. A less conventional MMU uses a Z80 slave processor

^c e.g., due to the mapping scheme restrictions, the data, the stack, and the *udot* are allocated separately on a system with a 68451.

to handle all memory allocations and management. Most of these MMUs have the advantage of being simple and easy to set up by the kernel. However, some have disadvantages. Those with few registers have to be reset more often when switching processes. This might not be a problem when few processes are running, but would be more noticeable with many users. In some cases, the scheme is so simple that it is difficult to implement certain kernel features or functions requiring additional registers to map in more than the usual text, data, and stack area—such as the `phys` system call, shared text, and shared data. Severely restricting the mapping and the size of the kernel virtual space also creates a handicap when the kernel is accessing user area. For those with a huge number of registers, the kernel might require additional logic to set up only the necessary registers instead of setting all of them up when switching processes.

In general, MMUs with a limited number of registers are more suited to a swapping kernel, but are less suited to a virtual memory kernel. In the case of the slave processor MMU, which handles allocation as well as memory management, the system can turn into a virtual memory system without requiring any modifications to the kernel itself. It only requires changes in the MMU.

CONCLUSION

The amount of work done by the UNIX kernel depends on how the hardware translates virtual to physical address. A simple MMU allowing sufficient mapping registers is more desirable than a complex one requiring more effort from the kernel software. A simple design also expedites the process of porting the UNIX kernel. The overhead we have seen in a swapping kernel is expected to be more pronounced in a virtual memory system, because of the complex record-keeping and more frequent reloading of registers required. When designing a MMU, one should find a simple scheme that efficiently performs the frequent basic operations before considering fancier features for supporting virtual memory. Even though the affect of the MMU logic on system performance as a whole might not be significant, a well designed MMU does mean less work for the kernel. Using an already proven MMU instead of designing new ones, OEMs can also take advantage of the debugging and refining others have done.

Techniques for Debugging XENIX Device Drivers

Paresh K. Vaish & Jean Marie McNamara

Integrated Systems Operation - North
Intel Corporation
5200 NE Elam Young Parkway
Hillsboro, Oregon 97123

Path: hplabs!intelcal!omsvax!pkv or hplabs!intelcal!omsvax!jean
Phone: (503)681-5236 or (503)681-5481

ABSTRACT

Today most microcomputers and segments of the minicomputer industries are migrating toward non-proprietary operating systems. Many commercial customers no longer want to be locked into one company's proprietary software products. Intel's systems group has adopted the XENIX+ system as its standard operating system for the commercial marketplace. The XENIX system is Microsoft Corporation's direct port of AT&T UNIX++ with value added enhancements. UNIX systems have become a standard for commercial microcomputers. The most important reason for this success is the XENIX/UNIX open system concept. Customers can purchase application packages from a variety of vendors. They can migrate to new hardware technologies while protecting their software investment. Peripheral and board manufacturers can add new controllers because the XENIX system provides a means to easily add new device drivers to the operating system.

This paper is intended to help the novice XENIX device driver writer in the process of debugging a driver. Debugging a device driver is often a tedious task as the driver runs as part of the XENIX kernel and not as an application program. In addition, debugging the hardware interface a driver provides can be a non trivial task.

The use of several tools for debugging device drivers is described. These include a debug monitor and XENIX utilities such as **adb**, **nm**, **cc**, **tprint**, **ctags**, and **lint**. The use of several hardware/software tools is also discussed. The paper provides a troubleshooting guide which aids in the detection and correction of problems like race conditions, timing problems, and hardware interfacing problems. It concludes with a list of suggestions/warnings for driver developers.

+XENIX is a trademark of Microsoft Corporation.

++UNIX is a trademark of Bell Laboratories.

1. Introduction

This paper presents a brief description of the various tools/techniques the device driver writer should keep in mind while debugging a XENIX# driver. After discussing the techniques, a troubleshooting guide for device drivers is provided which presents problems and approaches towards their solution. The paper concludes with a discussion of mistakes commonly made by device driver writers in the XENIX environment. The authors recommend that the reader refer to **Writing Device Drivers for XENIX Systems** presented by them at the January UNIFORM conference prior to reading this paper as this paper assumes some familiarity with XENIX terminology.

2. Techniques Available to Device Driver Debuggers

Below we present some general techniques and tools available to the XENIX device driver writer for debugging device drivers. Later, in section 3, we show how these techniques may be used to debug some common problems.

2.1. Debug Monitor

An important tool available to the device driver writer is an assembly language listing of the code produced by the C compiler along with a cross reference table which shows where in system memory the various text and data objects utilized by the device driver, and other text and data objects in the kernel are stored. The XENIX C compiler, `cc`, when executed with the `-S` option, provides the assembly language listing in a file with the same name as your driver file but with a `.s` suffix. The utility `nm`, when executed with the kernel image as an argument, provides an alphabetical reference table. Using the utility `sort`, the driver writer can produce a sequential listing (by memory location) of the variables*. Note that each procedure will have an underscore character prefixed to its name in the listings produced by `nm`. The sequential listing is useful if the processor is executing an instruction in memory and the software developer would like

```
0150:07e8 T _clkstar 0150:4ba4 T _ecclse 0150:b810 T _getgid
0150:c1c0 T _clocal 0150:3478 T _ecinit 0150:258a T _getint
0150:902c T _clock 0150:36fa T _ecintr 0150:05be T _getisr
0150:c522 T _close 0150:4e0e T _eciocli 0150:25da T _getlong
0150:94f4 T _closef 0150:3fa4 T _ecopen 0150:d0b8 T _getmdev
```

Figure 1: Alphabetic listing from `nm`

```
0150:07ae T _splbuf 0150:1292 T _nosys 0150:3234 T _xccdec
0150:07b2 T _spl7 0150:12a0 T _nullsys 0150:3288 T _xumount
0150:07c3 T _splx 0150:12a8 T _joint 0150:32b6 T _xrele
0150:07ce T _idle 0150:1338 T _strayin 0150:32e4 T _xuntext
0150:07db T _waitloc 0150:134e T _usermod 0150:333a T _xmaptex
```

Figure 2: `nm` listing sorted by memory location

#XENIX is a trademark of Microsoft Corporation

* `nm /xenix | pr -3` #produces the alphabetic listing in 3 columns (see Fig 1 for part of this)

`nm /xenix | sort | pr -3` #produces the memory sequential listing in 3 columns (see Fig 2 for part of this)

For example, in Figure 1, we show part of an alphabetic listing produced by `nm`. As shown, the XENIX procedure `close` starts at address `150:c522`.

to know which high level language procedure is being executed.**

Monitoring processor activity is done using a debug monitor. For example, consider **System Debug Monitor 286 (SDM 286)**, a monitor that executes on Intel's XENIX Systems which provides the basic features required of debug monitors.

The monitor typically resides in PROMs on the processor board. It is entered upon power up of the system or, more importantly for the device driver writer, by pressing an interrupt button located on the exterior of the micro computer system. When this is done, the user is placed into the monitor and he/she may inspect the various processor registers and any part of memory symbolically. (i.e. The monitor will interpret memory values into instructions for the user). A prompt, ".", (dot) is provided. A typical sequence in debugging a device driver includes: "single stepping" the driver code using the monitor to ensure the code is functioning as expected. To do so, the following steps should be followed: (Note: Analogous commands are available in most debug monitors)

1. Obtain a list of addresses of all procedure variables using "nm" described above.
2. Type in a shell command line which will result in executing driver code, but instead of hitting carriage return resulting in command execution, push the interrupt button on the system. This results in entering the debug monitor.

For example:

```
$ tar tvf /dev/df0 <press interrupt button>      #read floppy disk
```

The command "g" to the monitor, followed by a carriage return to the shell will result in the execution of the command line. However, it is much more informative to execute to a certain point, then check if all variables are set to what they should be at that point, and then continue. To do this, check the nm alphabetic listing and get the address of a procedure to execute to. Then type in:

```
. g, <address>
```

Where <address> is the logical address of the procedure at which you want to interrupt instruction execution (set a breakpoint). The CPU will execute instructions to the instruction at the address specified and the monitor will be entered. A prompt will again be provided.

3. Take a look at the code which will be executed once you resume execution of commands by typing in, for example:

```
.20 DX <address>
```

where <address> is the address found from the namelist (nm) above. This displays the next 20 instructions which the processor will execute once your program is resumed.

4. "Single Step" (execute) through instructions by typing

```
.N, <return>
```

The next single instruction will be displayed. To single step, simply type ", "

```
, <return>
```

This will allow single instruction execution for as many times as you type the comma.

5. At some point it may be desirable to check what values are in registers, on the stack, or in memory. Displaying the register contents as well as determining the SP address may be accomplished by typing:

```
. X <return>
```

** For example, if the CPU was at executing an instruction at 150:12ab, looking at the sequential listing from nm in Figure 2, you could tell that it was executing code in the XENIX procedure ioint.

6. The stack may be examined by using the contents of the stack pointer address, and successive lower addresses (if the stack grows down toward low memory) using the "display (D)" command. The entire stack for the current procedure extends from the base pointer address (bp) to the stack pointer address (sp). Immediately below the bp is the return address for the procedure the CPU was executing before the current procedure was entered.

7. If a register or memory address contains an incorrect value, and this is interfering with further execution of code, the "examine" (X) command can be used to modify the register's value. To change the contents of the AX register to 5:

```
.X AX=5
```

8. To stop single stepping, simply hit carriage return. To resume normal code execution until another procedure is executed, simply "go" to its address listed in the **nm** table. Some monitors similarly allow users to execute instructions until a certain variable in memory is referenced or changed.

9. Finally, normal code execution can be resumed by using the "g" command.

2.2. The **adb** utility

If your device driver is being used in conjunction with some software that runs as a user process (for example, a disk driver whose services are being used by the utility **tar**), then the **adb** utility may prove useful. **adb**, a debugging tool which runs under XENIX as a user process allows the user to set breakpoints in user code and to examine core dumps produced by user programs. While a tutorial on **adb** is outside the scope of this paper, it should be noted that **adb** shows assembly language code and hence an assembly listing of the user process code would be handy when using it. **adb** will only be useful for debugging problems with the software that uses the device driver. For example, data being placed into a parameter buffer (which will, in turn be passed to the driver) in the wrong order can be detected using **adb**. Programs which dump core when executed can also be examined using **adb** to determine the cause of the problem.

2.3. The **ctags** utility

The utility **ctags** can be used to create a cross reference table of high level language procedures. If your driver consists of several procedures, you can use **ctags** to create a table that states which procedures are defined in your device driver, where they are defined, and what the calling sequence for those procedures is. This tool is mostly useful for tracking down undefined procedures, locating driver procedures quickly, and checking calling sequences of procedures. The device driver writer may consider using **ctags** to generate a cross reference table of all kernel procedures in order to help him/her quickly locate the various kernel procedures drivers may invoke.

2.4. Configuration Check

Often a device driver may not execute correctly (or at all) not because of bugs, but because it is configured into the kernel incorrectly. There are three configuration files which are important: **/sys/conf/master**, **/sys/conf/xenixconf**, and **/sys/cfg/cxxx.c** where **cxxx.c** is your driver specific configuration file. If the new kernel will not boot and the hardware checks out correctly, there may well be a problem with the way the driver is configured into the kernel. Improperly initialized configuration pointers can cause severe problems including destruction of kernel data. If an incorrect configuration is suspected, the first obvious step is to study the configuration file **cxxx.c** to make sure it contains what was intended. Then if the kernel boots, use the monitor to

check that the variables and pointers set up in the configuration file are correct immediately after the kernel comes up. If the kernel will not boot or it boots but can't talk to the board, the board address in the configuration file should be checked against the actual board addresses. The **master** file should be carefully checked whenever a configuration problem arises to verify all values for the device are correct. Make sure the device is included in the new kernel by checking the **xenixconf** file. Check the **/dev** directory to make sure all device nodes necessary are present, the major numbers correspond to those listed in the **master** file, and the minor numbers are as the driver expects them. Look at the **c.c** file to confirm that the major/minor numbers are what they should be.

A common problem that will prevent the developer from even creating a kernel is using the incorrect prefix in the **master** file for his driver procedures with the result that procedures defined in the **c.c** file are not in the driver at all.

2.5. The **tprint** procedure call

Sometimes while the user is executing device driver code, a bug is seen, but when "printf's" are included to print out the value of various variables, the bug ceases to occur. This is a classic symptom of a timing problem where the time delay introduced by the **printf**, which requires i/o, allows execution to progress normally. There are two ways to handle this situation. First, it may be the case that the hardware actually requires a delay at this point. Check the hardware manual against what is being done in the code to see if this is the case. Often, however, while the bug at this point disappears, the driver has other problems at a later stage. The hardware doesn't need a delay.

This leaves the user examining the code and trying to determine what is going on. This may be done by executing the code with breakpoints, using the monitor to check the value of variables. Since no delays are introduced, the bug may show up and, hopefully, the reason for it. However, sometimes, using the monitor itself introduces a delay enough to eliminate the bug. In such cases, the XENIX kernel procedure "tprint" can be used instead of "printf". **printf** introduces delays because it is writing to a console. **tprint** outputs to a buffer which is much faster. After execution, the buffer can be examined to see what the variable values were. Note, however, that the buffer used by **tprint** is only 1K bytes long.

2.6. The **lint** utility

Most C compilers do not do strict type checking or much verification of semantic correctness. All device driver writers should use the "lint" utility as a preventive measure. **lint** can catch some potentially disastrous situation, especially involving invalid pointer assignments and incorrect parameter passing. While many extraneous error messages may be generated by **lint**, any real errors it does catch will save the developer much time later.

2.7. Software Tracing and Debug facilities in XENIX

Using "printf" to display variable values is one of the most popular means of debugging any piece of code, and its use is also helpful in debugging device drivers. Using the C preprocessor, statements like the following may be inserted in the device driver code:

```
#ifdef DEBUGSW
    < insert debug print statements and other code here >
#endif
```

This would include the debug statements in your code conditional upon your defining `DEBUGSW` at the start of your driver. Alternatively, several degrees of debugging may be built into your driver. Thus, for example, in several procedures, you may add code which looks like:

```
#if DEBUGLEVEL > n
    < insert debug print statements and other code here>
#endif
```

where `n` is any number. Then, if the driver is compiled with `DEBUGLEVEL` defined to be some value `m`, only debug code which had `m > n` would be included. As a third option, developers may want to use pure C code to insert debug statements:

```
if (debuglevel > n) {
    < insert debug print statements and other code here>
}
```

The advantage here is that the global variable `debuglevel` could be modified by the user at run-time (using the debug monitor) to get more or less debugging information.

2.8. Diagnostic Tests

Some tools which are often useful when one is unsure that his/her hardware is functioning correctly are system and board level diagnostic tests. Many hardware boards come with on-board power-up diagnostics. Read the hardware reference manual for the board to determine the necessary jumper settings to execute these. If these tests pass, the CPU board may have a system confidence test in PROMs on board that will test your system for basic functionality. Finally, some systems come with special board specific diagnostic software that can be booted off a floppy or winchester device (assuming at least one of these works). These tests can diagnose the problem more accurately. For example, in the case of a bad hard disk, the diagnostic may tell the user which blocks are bad so that these blocks may be marked as `badblocks` using some utility.

2.9. In-Circuit Emulator

The in-circuit emulator is an extremely powerful but brute force approach to debugging your device driver. It is equipped with a probe which replaces the CPU and which performs all the functions the processor would normally perform. Using this tool, driver writers may see the exact sequence of instructions executed by the CPU (the emulator can typically store several hundred instructions in its memory) as well as set breakpoints on certain addresses being executed or on data values appearing on the system address or data buses, or in memory locations. The emulator also allows the user to single step the processor through the execution of instructions. This tool is especially useful when debugging problems involving interrupt handling, because the alternate tool, the debug monitor, needs to be programmed at a fixed interrupt level and thus has to have its own interrupt handling code. An in-circuit emulator is also useful for debugging device drivers which handle i/o to the console on which the debug monitor does its i/o.

2.10. Logic Analyzer

A slightly less complex investigation tool is the logic analyzer. This can be used to monitor data on a specific board or to monitor values as they are read into or output from given chips on a board. This is typically used to debug the hardware-software interface of the device driver. For example, the logic analyzer may be used to see if the board is receiving the parameters your software is passing correctly. Another use of the logic analyzer could be to monitor the interrupt line of the board to see if the board was actually generating interrupts as expected. These interrupts might not be noticeable if an oscilloscope was used to monitor the interrupt line as the duration of the interrupt cycle is very brief.

3. Trouble Shooting Guide

Below are some hints for approaching problems frequently encountered by device driver developers.

3.1. Problem: New Kernel Does Not Boot

Solution: When you create your new kernel, check its size using the **size** utility before you try to boot it. This utility provides you with the code size, data size and uninitialized data (bss) size of your kernel. Ensure that these values do not exceed the maximum sizes allowed for them by your machine architecture. If the kernel starts to boot but hangs after printing out a few messages, look at **dinit\$w** in the file **c.c** produced as part of your kernel generation process. Each routine in this data structure is called sequentially at boot time and each usually prints out a message. Most likely, the routine in this data structure corresponding to your driver is where the kernel is getting halted. If everything looks all right in the routines, pull out all new boards added to the system, reducing it to the minimum configuration and try again. Those boards may have been tying up a bus in your system. Another way to verify that you have no hardware problems is to boot a kernel that you know works (e.g. the kernel the system was shipped to you with).

3.2. Problem: Board/Device appears not to work when commands are issued to it

Solution: First, if possible, swap out your board and try a new one. If this board exhibits the same symptoms, you most likely don't have a bad board. Next, check the jumper settings on your board against those suggested in the hardware reference manual. If your device has any connections to terminals, Ethernet* cable, etc, verify that all the connections are good and locked in place. Then, try to use the debug monitor discussed earlier to verify that you are passing parameters to the board correctly. If they look correct, use a logic analyzer to verify that the board is getting the values that you intended. If they do not look correct, or nothing is being passed, you probably have device configuration problems. Check to make sure you added your driver correctly to the **master** file in **/sys/conf**. Then, verify that the file **/sys/conf/c.c** is being created correctly. Look especially at the data structures **bdev\$w** and **cdev\$w**. One other configuration issue you should be concerned with at this point is having correct device node major/minor numbers. Also verify that the node is of the correct type (block or character). Finally, consider all the data structures the driver initializes in the driver specific configuration file located in **/sys/cfg** and make sure you have no configuration problems there. If all this looks correct, contact the board vendor and discuss the scenario you are trying with him/her to ensure you understand the hardware interface correctly.

*Ethernet is a trademark of Xerox Corporation

3.3. Problem: Device Ties up Data/Address Bus of the System

Symptoms: The Data or Address bus is tied up when the main system processor is unable to execute any instructions at all. System Confidence Tests will not work and no i/o to or from the system is possible.

Solution: Try removing the new board(s) you added to your system and see if the problem disappears. If so, try another new board, verify the jumper settings, and see if this board works. If not, contact the board vendor and verify that the board is intended for use on your system bus. If the problem does not go away once the board you added is removed from the system, you have a bad system, a problem outside the scope of this paper.

3.4. Problem: Board works in some slots, not others

Solution: If your system bus is prioritized, the board is probably in the wrong priority slot. Check the bus manual for the correct position of the board. If, however, the bus is not prioritized, as in the case of MULTIBUS®, then you have a problem. First check the board's hardware configuration. If this is incorrect, the board could behave unpredictably. Next, verify that the firmware on the board is the correct revision and is installed correctly. Finally, check your bus' jumpering of priority levels, if any, to ensure it is correct.

3.5. Problem: Driver Software does not always function correctly (Race Conditions)

Symptoms: The problem disappears when delays are introduced; events occur in an unexpected sequence; problems which occur intermittently.

Solution: Try to isolate the code involved and decide what the critical timing events are. This is one of the most difficult and time consuming debugging tasks associated with device drivers, as many times the problem is not repeatable. Try to find a case that **is** repeatable, then use **tp**rint or the monitor to step through the code and check variable values. Introducing delays at strategic places may give more information as to where in the code the race condition exists. One may use the kernel routine **delay()** to do this. Shared code sections are highly suspect areas for race conditions. Make sure these areas are mutually excluded upon correctly. If you do have a case where the problem has occurred (for example, if your system has just issued a panic and your system is idling), try to break into the monitor immediately and study the various variables your driver uses and the stack and try to determine what sequence of events caused the problem.

Another frequently used technique to solve race condition problems is to form a hypothesis as to the sequence of events that could be causing the problem that is occurring and then using the tools described in Section 2 to prove or disprove this hypothesis.

3.6. Problem: Various data structures seem to be destroyed (Overwriting memory)

Solution: Typically, this results from bad or uninitialized pointers. Since the driver runs as part of the kernel, it is not restricted from writing wherever it pleases (whether intended or not!). Thus, the driver may destroy the file system on a disk or perhaps even wipe out parts of the kernel's code. An example of this problem is a case where a procedure declares a pointer, then proceeds to use it to store values in a structure without initializing the pointer to point to an instance of that structure. To correct this, the pointer(s) at fault must be isolated. The best way to do this is with the monitor, single stepping through the program, checking the various pointer

instance of that structure. To correct this, the pointer(s) at fault must be isolated. The best way to do this is with the monitor, single stepping through the program, checking the various pointer values as you go.

Less commonly, incrementing or decrementing a register may result in register overflow. If this is not an expected event, the symptoms described may result. This can be detected much in the same way an errant pointer is: using the monitor to step through the program, checking registers as you go. If the device interface contains a register, it is possible for it to overflow also.

Writing to the u-structure at interrupt time is not appropriate as there are no guarantees from the kernel about which process will be running at interrupt time. Thus, another processes' u-structure may be overwritten. This may result in problems for the driver process as well as another process. In general, it is important to remember a device driver is kernel code and any access of user space [user text and data is distinct from kernel text and data] must be done explicitly using special system routines. If this is not done, the driver may inadvertently overwrite system memory.

3.7. Problem: Driver sees spurious interrupts

Solution: Spurious interrupts are unexpected interrupts from any device in the system. If a spurious interrupt is observed, several things should be checked. First, many devices require that an interrupt be "cleared" in some way. If this is not done correctly, the hardware may act unpredictably and generate unexpected interrupts. If the interrupt handler invokes the `sleep()` procedure, unexpected system behavior may result as the handler is not a process. Many drivers have a variable that indicates whether they are expecting an interrupt and check this variable in the interrupt handler to help them decide if the interrupt is spurious. If it is, the drivers simply ignore this interrupt. If an unexpected interrupt occurs at some other level, there is a hardware problem in the system. The CPU board is the first suspect -- it should be swapped out. If the spurious interrupt level corresponds to a board in the system, the board should be checked by trying a new board.

3.8. Problem: Characters being sent to a terminal appear to be getting lost

Solution: This is a fairly general problem but one possibility in this case that is often overlooked is that the hardware (either the board or the terminal) may not be able to keep up with the baud rate which the device driver is trying to obtain.

4. Warnings

In this section, we present some warnings/suggestions that will help the device driver developer produce bug-free code.

- The kernel does not guarantee that the maximum kernel stack size normally available to processes will be available to the interrupt handling code at interrupt time. This is because interrupt handlers use the stack of the process that was executing when the interrupt came in. Thus, interrupt handlers, and procedures invoked by these handlers should use minimal data structures. Using large arrays, for example, which have to be stored on the process' stack, could lead to stack overflows and data being overwritten.
- Before doing any testing of the hardware/software system, developers should check all connections to the system from the board/device. This includes connections to terminals from

communication boards, connections to Ethernet taps from Ethernet controllers etc. Whatever connections can be locked together in hardware should be so locked. Finally, check the edge connectors on the new board to ensure that the connections to the system bus are not covered with any form of insulator.

- User programs cannot access kernel space. However, a device driver is not a user program; it is part of the kernel. As a result, device driver code has the capability of accessing any part of the kernel as well as the potential for overwriting user text and data. Thus, the driver writer needs to clearly understand the difference between kernel and user space.
- Static data structures used by the driver should have a prefix like 'd_' which will help to differentiate these data structures from kernel data structures of the same name. This is important since name conflicts could otherwise result.
- In C, if one tries to access a variable in a structure x, which was declared as part of another structure, y, and not in x at all, then some C compilers (cc), may well not object, and unpredictable behavior may result. Thus, all structure references, especially assignment of values to parameter blocks, should be carefully coded and checked.
- Interrupt handlers should not use the kernel procedure `sleep()` or invoke any procedure which invokes `sleep()`. Invoking the `sleep()` kernel procedure from the handler will, as mentioned earlier, cause unexpected system behavior.
- It is usually a good idea to compile driver code without optimization switches until it is entirely debugged. Optimize switches may produce code that has bugs which were introduced during the optimization process. The main reason optimizing code may cause problems is that optimizers may not handle the case where even though a variable does not appear to be changed for several lines of code, it is actually modified by the interrupt handler which might be invoked in between the execution of those lines of code.

User-Mode Development Of Hardware and Kernel Software

Robert P. Warnock, III

Fortune Systems Corporation
Redwood City, California 94061

ABSTRACT

As a general rule, the development of new hardware devices, operating systems drivers for those devices, and other new operating systems functions is considerably more difficult than the development of user-mode functions of similar complexity. Several factors contribute to this: hardware often doesn't work as initially expected (despite documentation); testing drivers and other kernel functions requires a very scarce resource — stand-alone time on the system; errors often leave the entire system hung or halted with no history trace, making crash analysis a challenge at best; the edit-compile-load cycle tends to be longer and more complex; and a logic analyzer is seldom the most convenient diagnostic tool.

A set of techniques or "tricks" are presented, with examples of their application. While each one may be "obvious" by itself, and not particularly related to the others, together they illustrate a common principle and general method. The principle is that of separation of concerns, together with addressing those concerns in the proper order. "First make it work correctly; then make it work well while remaining correct." The general method is to do the development in user-mode software, using minimal "hooks" to make this possible. Then, after the functionality has been demonstrated and the critical algorithms debugged, the software is "ported" to kernel mode as necessary to attain the required performance goals.

Other authors [Holt] [Wulf] have suggested, in fact, that the "kernel" of an operating system should be quite tiny (a few hundred lines of assembler), and that ALL of what one normally thinks of as the "operating system" should be run in user-mode, including device drivers, file systems, and schedulers. Unfortunately, most of us do not have the freedom to make major modifications to our operating system environment (typically UNIX† of some flavor or other). The examples given demonstrate that, at least during ini-

† UNIX is a trademark of Bell Laboratories.

[Holt] R. C. Holt, *Concurrent Euclid, The UNIX System, and Tunis*, Addison-Wesley, 1983

[Wulf] William A. Wulf, Roy Levin, and Samuel P. Harbison, *HYDRA/C.mmp*, McGraw-Hill, 1981

tial development, it is possible to obtain the benefits of the "user-mode style" even though the production version may be completely traditional in structure.

The development projects used as examples took place at Fortune Systems between Summer 1982 and Summer 1984, and include:

1. A byte-parallel file-transfer link was implemented between a DEC VAX-11/780 and a Fortune Systems 32:16. The VAX driver was developed in user mode using /dev/kUmem to access the hardware. The 32:16 driver was developed in user mode using the "sysphys" feature (UNIX Edition 7 "phys(2)" call) to map the user addresses to the hardware. After the file-transfer application was completely functional, the VAX driver was moved to the kernel, with a 25-fold improvement in performance. (The 32:16 driver was left in user-mode permanently.)
2. A communications co-processor for the 32:16 was debugged using user-mode software (again using "sysphys"). When the UNIX driver was being debugged, host-resident user-mode code was used to mimic the co-processor application on the one hand, while making calls to the driver and comparing the results on the other. A similar procedure was used in developing a bit-mapped graphics controller and a parallel-I/O co-processor.
3. A set of library subroutines was written to allow user-mode emulation of (proposed) new operating system calls. When the "system call" was invoked, instead of entering the normal (kernel-mode) system call handler, a call-request packet was passed through a "pty" to a daemon program which emulated the call and passed a "return value" packet back through the pty. Packet types were provided to allow the daemon to read and write the client process's address space (as the kernel would have been able to do).

This facility was used to develop a network "socket" mechanism (similar to 4.2bsd sockets). A "network line discipline" was implemented using ordinary terminal ports as network devices. After the internet router and network line discipline were completely functional running in user mode as a system-call emulation daemons (including actually transmitting packets over a multi-host net), they were "ported" straightforwardly into the kernel.

4. In the previous hardware examples, the physical device had its interrupts disabled when driven by the user-mode driver, so as not to crash the unmodified naive kernel with unexpected interrupts. (The user-mode drivers used either busywait-polling or sleep-polling for synchronization.) Similarly, DMA operation was not possible.

In developing a local-area network interface, it was necessary to utilize both of those features. A slight kernel modification was made to reserve a block of physical memory which the kernel would not use. User-mode library routines were provided that (1) allowed allocation of that memory area to DMA

operations (the results of which were then examined with "/dev/mem" or "sysphys"), and (2) allowed run-time installation of minimal interrupt-service routines (using "pre-compiled" templates) which merely stored the device status in a mailbox and cleared the interrupt (the user-mode driver polled the mailbox, rather than the hardware).

Again, the device driver was not "ported" to kernel mode until the hardware had been completely checked out, the device driver algorithms were debugged, and the sample application programs had demonstrated end-to-end functionality.

Several examples have been given of developing what is normally considered "kernel mode" software in user mode. While these examples are not likely to apply directly to other environments, it is hoped that implementors will be encouraged to consider the "user-mode style" when planning future kernel-mode software development projects.

**Relating Benchmarks to Performance Projections
or
What do you do with 20 pounds of benchmark data?**

**Gene Dronek
Aim Technology
Santa Clara, CA 95051**

Abstract

For reasons unknown, UNIX operating systems seem to invite benchmarking measurements. So much so, it has become popular for UNIX journals to include benchmark-of-the-month columns. But can you really use these programs to justify purchase decisions, or is benchmarking merely an interesting exercise?

At Aim Technology, we have been researching how to construct performance models based on raw benchmark data. As an experiment we built a linear-combination model that projects a single "figure of merit" from 1 to 50 columns arbitrary benchmark data. The way we implemented it, you request a projection by specifying percentage mixes of applications, such as 10% word-processing plus 90% compiling.

The model worked so well, we exercised it generously on a variety of hardware configurations, but quickly were buried once again with too many benchmark numbers. Then, at a Palo Alto UNIOPS meeting, we heard about Kiviat diagrams, which look much like starburst printouts. From our linear model we discovered printing Kiviat diagrams could easily show as many as 50 performance factors for 20 machines simultaneously on one diagram.

We will present implementation details for the linear model and how to use it on your own projections. We will also show several interesting charts and Kiviat diagrams comparing well-known systems.

UNIX* System V and 4BSD Performance

Jeffrey P. Lankford

AT&T Bell Laboratories
190 River Road, Summit, NJ 07901

ABSTRACT

This paper characterizes the performance of AT&T System V Release 2 and 4.2BSD developed at the University of California Berkeley. These systems are also compared with their respective predecessors, System V and 4.1BSD. All systems were measured on VAX† 11/780 hardware.

1. INTRODUCTION

Previous benchmarking efforts have either focused on characterization of individual components^{[1] [2]} or have attempted to generalize system performance based on results from a small collection of commands.^{[3] [4]} Comparison of results from these and similar approaches is generally inconclusive and sometimes contradictory.

The present research synthesizes both approaches. While recording system wide and per-process resource use, a workload imitating multiple users engaged in program development provides an estimate of capacity in terms of process throughput. Various software components are then measured individually, including the C compiler and library, many system calls as well as disk and terminal I/O. The results provide enough evidence to predict performance consequences of system release upgrades for a variety of applications.

Although significant timing differences between System V and 4BSD are observed in measurements of several components, ability to support program development is roughly similar. Results show that overall improvement of System V Release 2 over System V is attributable to enhancements of some commands and C library functions. Although the performance of some important file operations improved, decline in performance of several other components resulted in an overall capacity reduction for 4.2BSD from that of 4.1BSD.

2. METHOD

2.1 Hardware and Software Measured

Effects of software differences are isolated by using the same hardware for all tests, based on a VAX 11/780 processor without any auxiliary floating point processor.¹ Primary store consisted of 2 Mb on a single memory controller, while secondary store consisted of three **rp06** disk drives attached to a single MASSBUS. Terminal multiplexers included eight DZ-11 and four KMC-11B auxiliary processors on a single UNIBUS.²

Like other software, benchmark execution time depends on the underlying hardware. Moreover, test results from the system releases investigated may be affected to different degrees when hardware is changed. For example, although effects of 4.2BSD tunings to take advantage of timing properties of different disk storage hardware are known,^[5] behavior of other systems is not established. As another example, the DH-11 terminal multiplexer and its emulators enjoy widespread use under

* UNIX is a trademark of AT&T Bell Laboratories.

† VAX, MASSBUS and UNIBUS are trademarks of Digital Equipment Corporation.

1. Floating point computations occur only during the workload benchmark and then only in insignificant amounts.

2. Since KMC microcode for terminal handling is not provided, the KMC-11B processors were not configured into 4BSD systems.

4BSD but not with System V. Although it provides DMA capability that the DZ-11 does not, CPU involvement is required for cooked mode processing, whereas the KMC-11B under System V performs this processing by itself. Hence, 4BSD terminal handling performance using the DH-11 is expected to be better than that reported here with the DZ-11. In addition, although the amount of memory configured might be too little for a typical timesharing installation, on the workload test symptoms of memory shortfall were generally exhibited only by 4.2BSD. Except for the KMC-11B, hardware selection was restricted to that supported by all systems evaluated.

The system releases tested are: System V, System V Release 2, 4.1BSD and 4.2BSD. Tampering with the official system releases was kept to a minimum, but included fixes to known 4.1BSD bugs. A version of `cpio(1)` was ported to 4BSD systems and 4BSD `nroff(1)` was modified to permit processing of `mm(7)` macro formatted files.

2.2 System Configuration and Tuning

The environment is constructed so that system resource use during the workload benchmark is similar to that found at program development installations within AT&T Bell Laboratories. The tunable parameters and the file system layout used are given in the Appendix.³ All tests were conducted in single user-mode with `cron(1M)` and kernel profiling turned off to minimize background load. In addition, for all tests except the workload benchmark, process accounting was turned off. The Bourne shell, `sh(1)`, was used for all tests and had the `PATH` variable set during the workload benchmark so that the `fork(2)/exec(2)` ratio would be about three-to-one. Workload tests were performed both with and without sticky bits set on all the commands used.

2.3 Benchmark Techniques

The characterization tools and techniques are those used at AT&T Bell Laboratories to monitor the impact of software changes on performance.

The workload benchmark measures process throughput as a function of applied load, which is generated by concurrently executing an increasing number of standardized shell scripts.^[6] Script composition is based on CPU use from field site process accounting data, but does not include networking or system administration commands or other commands not distributed with all systems, except as already noted. Order of command execution within each script is varied to avoid synchronization problems, but no delay is interspersed between command invocations. Although not all commands executed are individually CPU bound, when several job streams are running simultaneously, the workload overall is CPU bound. Terminal I/O does not occur with this test, but is measured directly using another benchmark.

The C language benchmarks are a collection of seven programs, each about one hundred lines of C source code, intended to exercise basic C language capabilities.⁴ The code contains no I/O requests, nor are any C library routines referenced except for the subroutine register save/restore mechanism. Further, no system calls are made.

The C library measurement strategy involves repeating specified functions in a loop to obtain measurable execution times.⁵ Three categories of operations are examined: string functions, conversion and format I/O and the standard I/O library. Dependent parameters such as string length, conversion and data type as well as transfer size are varied to cover a wide range of usage.

3. With the above hardware, the automatic software configuration of 4.1BSD resulted in too few buffers allocated. Therefore, 4BSD default configuration was circumvented to establish a base nearly identical to System V.

4. The range of computations performed include: A doubly recursive function call, word sort, Quicksort, terminal handler driver, symbol table insertion, buffer release and statistical calculation.

5. The tight loops in which these short library and kernel code segments execute result in a higher than common cache hit ratio, yielding correspondingly optimistic time values. Performance comparisons are valid, since different systems running on the same hardware are affected to about the same degree.

The system call benchmarks similarly repeatedly invoke a primitive operation to obtain a measurable elapsed time. For operations with associated CPU idle periods, a "cycle soaking" process is run at low priority in the background to account for such periods. In these cases, CPU consumption equals the elapsed time minus the soaked time. The *fork(2)* tests are conducted both with and without program activities that cause writing of the child data segment following the operation, with the former case ensuring that a second copy of the data is actually made. For file system operations requiring inode table searches, the position of the target entry in the inode table is controlled by opening dummy files before opening the target file.⁶

A slightly different technique is employed to measure disk and terminal I/O. For these tasks, multiple active processes are used to sustain high use of the CPU and I/O devices.⁷ To transfer many disk blocks without having to access indirect blocks of large files, each process opens ten 8 Kb files and alternates transfers (reads or writes but not both simultaneously) round-robin fashion among the open files. When the file pointers reach the end of the files, they are reset to the beginning and this procedure is repeated many times. This scheme tends to defeat attempts by the system to cache disk blocks in memory (and read-ahead) and results in one physical I/O for each logical I/O. Terminal handling tests involve initiating one process per line that repeatedly writes (output) or writes then reads (loop around) a specified byte stream on the previously opened and set terminal line.⁸ Terminal tests are executed for both *raw* and *cooked* modes, for a variety of transfer rates (110 to 9600 baud) and over a range of transfer sizes (1 to 512 bytes) to determine the maximum realizable transmission bandwidth when the CPU is devoted exclusively to terminal handling. Only twenty lines were active during these tests and they were distributed over the available I/O ports.

3. RESULTS

3.1 Workload Benchmark

Figure 1 shows relative process throughput as a function of applied load with sticky bits set.⁹ In general, throughput increases rapidly to a peak value early in the curve, then gradually declines. Results without sticky bits are similar but throughput is lower, although 4BSD systems are less affected (partly because text pages are effectively cached in memory by the paging mechanism).

Peak throughput values extracted from Figure 1 are provided in Table 1. This shows a ten percent increase in System V Release 2 over System V. Capacities of System V and 4.1BSD are nearly equivalent. There is a twenty percent decline in 4.2BSD performance relative to 4.1BSD.¹⁰

6. Since for all systems examined the inode table is now hashed, this control is no longer important.

7. Where the CPU is not fully busy, time consumed by the cycle soaking process is used to project maximum rates by linear scaling.

8. A "null terminal" device sustains appropriate RS-232 voltages and circulates data when required (loop around).

9. 4.2BSD measurements beyond twelve scripts are not presented, because above this load *nroff(1)* processes terminated prematurely with an "/usr/bin/nroff: too big" diagnostic.

10. Results obtained for 4.1BSD using the automatic configuration tuning are ten percent lower than those in Figure 1 and Table 1 because of a lower buffer hit ratio. The 4.2BSD default configuration doubles the number of buffer pages available; hence, sites upgrading to 4.2BSD and retaining use of the automatic configuration mechanism should experience a more moderate capacity decline.

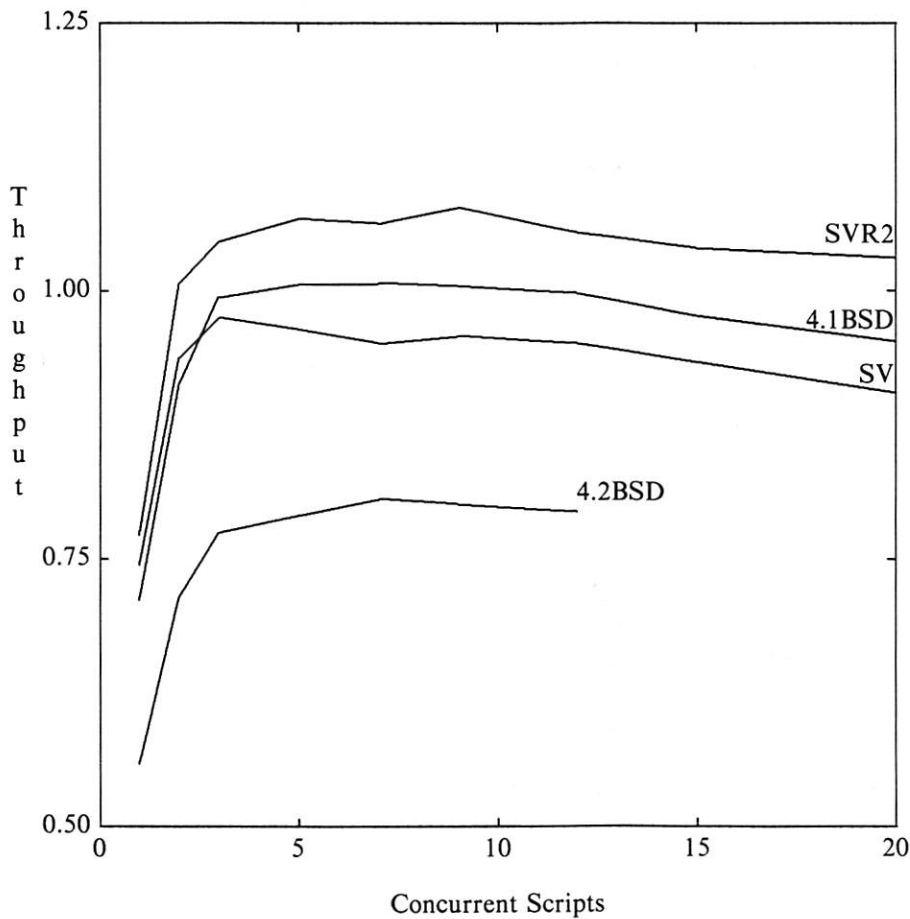


Figure 1. Workload Relative Throughput (jobs/min)

	AT&T	BSD	
SV)	0.98	1.01	(4.1
SVR2)	1.08	0.81	(4.2

TABLE 1. Peak Workload Rate (jobs/min)

System monitoring tools, such as `sar(1)` or `vmstat(1)`, are invaluable for assessing the workload results. In general, throughput increases until CPU idle time is eliminated, then tapers off as the buffer cache hit ratio declines and process switching increases. User-to-sys CPU utilization ratio provides a rough indication of kernel efficiency; System V and System V Release 2 exhibit a 50/50 split when the CPU is fully used, while 4.1BSD spends proportionally less time in kernel mode (55/45) and 4.2BSD expends a greater portion in the kernel (45/55). Rates for process switches, system calls and disk transfers are similar for all systems. Also, the number of disk transfers are roughly balanced among the three drives. However, while System V, System V Release 2 and 4.1BSD show practically no swapping or paging activity, 4.2BSD exhibits considerable paging activity even under light loads. Also, the amount of free memory is lower and active virtual memory is greater for 4.2BSD, because of the larger `bss` segment sizes of C programs.

Process accounting CPU times, summarized in Table 2, show the effect of command development efforts and help explain workload results. Data for 4.2BSD are not available, since the process

accounting package as distributed does not work properly. Most process CPU times are similar across system versions, but notable exceptions include: **as(1)**, **cat(1)**, **ed(1)**, **nroff(1)**, **pr(1)**, **sh(1)**, **spell(1)/spellprog** and **who(1)**. **Echo(1)** and **pwd(1)** are built into the System V Release 2 shell,¹¹ hence CPU times for these operations are inseparable from the **sh(1)** time.

<i>Process</i>	<i>SV</i>	<i>SVR2</i>	<i>4.1BSD</i>
as,cc,ccom,cpp,ld	1.1	1.2	1.0
cat,pr	1.2	1.1	0.8
col,nroff	7.8	6.2	6.8
cpio	1.8	1.8	1.7
diff	0.6	0.6	0.7
echo,pwd,sh;[2.1	1.6	2.5
ed	1.5	1.4	1.2
find	0.7	0.7	0.8
ls	0.2	0.3	0.2
make	0.2	0.2	0.2
mkdir,rmdir	0.6	0.6	0.6
ps	0.3	0.3	0.3
rm	0.6	0.6	0.7
spell;spellprog	0.3	0.3	0.8
OTHER	0.8	0.8	0.7
Total	19.8	17.7	19.0

TABLE 2. Process CPU Time at 20 Scripts (min)

3.2 Component Level Tests

Normalized mean object code execution speeds are shown in Table 3, with faster program execution implied by larger numbers. In general, differences in object code execution speeds are small, since there are no significant differences in the assembler code generated before optimization. The difference between 4.1BSD and 4.2BSD results is experimental noise.

	<i>AT&T</i>	<i>BSD</i>	
SV)	1.00	0.96	(4.1
SVR2)	1.00	0.95	(4.2

TABLE 3. Normalized C Language Execution Speed

Execution times for several representative C library functions are listed in Table 4. Except for the *fread(3S)* operations (directed to in-memory buffers),¹² all systems exhibited similar results. System V *fread(3S)* of large chunks is about four times faster than 4BSD because this function reads directly from the buffer rather than loops on *getc(3S)*. Although string operations are unchanged, conversion and format operations as well as standard I/O library functions improved slightly for System V Release 2.

11. In addition to other features gleaned from **csh(1)**, like path name hashing.

12. Since all data is manipulated in memory, 4.2BSD C library enhancements for buffer filling from or flushing to variable block size file systems are not shown by these tests.

<i>Operation</i>	<i>SV</i>	<i>SVR2</i>	<i>4.1BSD</i>	<i>4.2BSD</i>
strcpy 1 b	0.04	0.04	0.03	0.03
strcpy 64 b	0.09	0.09	0.15	0.15
atoi	0.06	0.06	0.06	0.06
fprintf (long)	0.32	0.24	0.28	0.29
sprintf (long)	0.32	0.23	0.28	0.29
fread 1 b	0.09	0.09	0.05	0.05
fread 1 Kb	2.4	2.3	11	11
fread 4 Kb	9.3	9.1	45	44

TABLE 4. C Library Function CPU Time (msec)

Table 5 provides execution times for a representative selection of system calls. In general, results for all systems are close to one another and rarely is the difference between any two systems greater than a factor of two. There is no important difference between System V Release 2 and System V, since no significant kernel changes were made. Note that almost all operations under 4.2BSD are slower than the same operation under 4.1BSD.

<i>Operation</i>	<i>SV</i>	<i>SVR2</i>	<i>4.1BSD</i>	<i>4.2BSD</i>
<i>getpid</i>	0.18	0.18	0.18	0.19
<i>lseek</i>	0.33	0.30	0.57	0.65
<i>chdir</i> ''	1.2	1.3	1.2	1.6
<i>open/close</i> 'clsfile'	2.5	2.5	2.4	3.0
search path 3 lvls.	5.9	6.1	6.0	6.4
search dir. 32 pos.	3.0	3.3	4.2	4.1
pipe 1 b	2.6	2.6	2.1	2.6
pipe 1 Kb	3.9	4.0	3.5	4.0
pipe 4 Kb	11	11	11	9.7
read b. cache 1 b	0.94	0.85	0.64	1.0
read b. cache 1 Kb	1.8	1.7	1.5	1.8
read b. cache 4 Kb	5.5	5.2	5.0	4.1
<i>fork/exit</i> 16 Kb - touch	29	29	49	61
<i>fork/exit</i> 16 Kb - not touch	29	29	29	37
<i>exec</i> 16 Kb bss	23	23	24	30
<i>exec</i> 16 Kb data	37	39	28	33
<i>exec</i> 512 b arglst	62	64	55	77
read disk 1 b	4.4	4.2	3.7	5.4
context switch	0.71	0.75	0.57	0.65

TABLE 5. Kernel Operation CPU Time (msec)

Disk read throughput for several transfer sizes is shown in Table 6; disk write throughput is similar but generally lower. 4.2BSD supports an 8 Kb block file system whose throughput increases nearly linearly with block size as per-transfer overhead becomes proportionally smaller. Had the UNIX buffer caching strategy not been defeated, 4.2BSD results for all transfer sizes would be the same as the 8 Kb result. System V, System V Release 2 and 4.1BSD support only 1 Kb block file systems and they show throughput nearly equivalent to one another.¹³ For these systems, differences in

throughput for different transfer sizes are experimental noise resulting from sequence variation of blocks on the free list.

<i>Read size</i>	<i>SV</i>	<i>SVR2</i>	<i>4.1BSD</i>	<i>4.2BSD</i>
1 Kb	70	70	80	40
2 Kb	60	60	70	80
4 Kb	60	60	80	120
8 Kb	70	60	80	180

TABLE 6. Peak Disk Read Rate (Kb/sec)

Peak terminal throughput for output (9600 baud) and loop-around (1200 baud) is shown in Tables 7 and 8, respectively. Although terminal handling for System V Release 2 and 4.2BSD was not measured, results should be similar to their predecessors, since there are no significant changes in this area for either system.¹⁴ Using the KMC-11 for output processing, System V rates are impressive. The 4.1BSD output results illustrate the effectiveness of changing the driver to compensate for lack of hardware DMA by processing more than one character per interrupt. Loop around rates are about ten times less than output rates.

<i>Mode</i>	<i>SV+KMC</i>	<i>SV-KMC</i>	<i>4.1BSD</i>
raw	40	5.6	16
cooked	40	2.9	8.1

TABLE 7. Peak Terminal Output Rate (Kb/sec)

<i>Mode</i>	<i>SV+KMC</i>	<i>SV-KMC</i>	<i>4.1BSD</i>
raw	4.2	1.7	0.9
cooked	2.4	1.1	1.6

TABLE 8. Peak Terminal Loop-Around Rate (Kb/sec)

4. DISCUSSION

The results presented lead to three primary conclusions: performance for all systems is comparable, 4.2BSD performance declined slightly relative to its predecessor and System V Release 2 performance improved somewhat relative to its predecessor.

The key finding is the general similarity in performance of System V and 4BSD, on both the program development workload and the component level tests. When primary store is not a limiting resource (as with this workload), swap based system performance does not necessarily differ from that of page based systems. Although 4BSD kernel code segments are tuned to avoid unnecessary

13. System V and System V Release 2 also support 512 byte block file systems, but throughput is two times lower than the 1 Kb file system.

14. Actually, 4.2BSD is expected to do better than its predecessor on the loop-around test, since configuration changes now enable the DZ-11 receiver silo by default.^{17]}

overhead (for example, data is not copied after a *fork(2)* unless touched), differences between kernel operation times are generally modest. The notable areas of dissimilarity are in I/O handling; 4.2BSD is superior by a factor of two in disk throughput for large transfers owing to use of larger size blocks, while System V is superior by a factor of two in terminal throughput because of off-loading of character processing to a front-end processor.

A surprising finding is the capacity decline of 4.2BSD relative to 4.1BSD. The magnitude of the decline shown by the workload benchmark is reflected by the decline in execution speed of many system calls. Although file system throughput increases as transfers are made using large blocks, throughput declines when using 1 Kb blocks. Hence, overall system performance partly depends on the degree to which programs use large block sizes for disk transfers.

The capacity of System V Release 2 increased over that of System V because of improvements in a few commands and library functions. This means that performance improvement depends on the degree to which the enhanced commands and libraries are used, so that improvement may not be as widespread as might be the case if kernel enhancements had been the basis.

Except for file system performance, the present results agree with those of the previous investigators cited. Although the method used here for file system throughput characterization is different from that used by McKusick, the two approaches produce similar maximum throughput results for 4.2BSD; however, the author obtained better performance results for 4.1BSD. This yielded a smaller difference between systems than that reported by McKusick.¹⁵

Applications exercising system components in a fashion significantly different from the program development workload may obtain a different overall result. In particular, where conditions of extreme memory shortfall and large process execution occur, advantages of paging over swapping should become prominent. This area might provide a fruitful avenue for additional performance characterization.

APPENDIX

Table A1 shows file system and disk drive usage with the workload benchmark. The swap area for System V was 9000 512-byte sectors at the end of the *root* file system, while swap areas for 4BSD were */dev/rhp0b*, */dev/rhp1b* and */dev/rhp2b*.

<i>Name</i>	<i>Use</i>	<i>AT&T</i>	<i>BSD</i>
<i>/</i>	standard: commands read from <i>/bin</i>	rp00	hp0a
<i>/usr</i>	standard: commands read from <i>/usr/bin</i>	rp01	hp0g
<i>/tmp</i>	standard: temporary files used by commands	rp12	hp1a
<i>/bck</i>	ten script directories	rp13	hp1g
<i>/ud</i>	directories used for running benchmark	rp24	hp2a
<i>/mnt</i>	ten script directories	rp25	hp2g

TABLE A1. File System and Disk Drive Usage

In Table A2, parameters with no corresponding equivalent on a particular system are represented by a dash; parameters enclosed by *[]* are used by 4.1BSD only, while those enclosed by *()* are unique to 4.2BSD. Note that about the same number of bytes are reserved for terminal handling queues, since *clists* are 64 chars and *nclist* structures are 28 chars. Also, since the size of *buffers*, *nbuf* and

15. An informal independent effort to reproduce McKusick's results confirmed the present findings.^[8]

<i>Parameter</i>	<i>AT&T</i>	<i>BSD</i>	<i>Parameter</i>
	--	32	maxusers
maxproc	75	75	MAXUPRC
hashbuf	64	--	
buffers	200	200	[nbuf],(bufpages)
calls	50	50	ncallout
inodes	350	350	ninode
files	300	300	nfile
mounts	18	18	NMOUNT
procs	160	160	nproc
texts	60	60	ntext
clists	200	460	nclist
swapmap	85	86	[nswbuf]
	--	100	(nbuf)
	--	50	(swbuf)

TABLE A2. Operating System Tunable Parameters

bufpages are all 1 Kb, the same amount of memory is used for disk buffering by all systems.

REFERENCES

1. Joy, W. Comments on Performance of UNIX on the VAX. Computer Science Division internal report, UCB, 1980.
2. McKusick, M. K., et al. A Fast File System for UNIX. CSRG at UCB, draft of March 11, 1983.
3. Tuori, M. A UNIX Benchmarking Tool. DCIEM Research Paper 82-P-23, June 1982.
4. Babaoglu, O. and W. Joy. Converting a Swap-based system to do Paging in an Architecture Lacking Page-reference Bits. CACM, December, 1981, pp78-86.
5. Kridle, B. and M. K. McKusick. Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX. CSRG at UCB, Revised July 27, 1983.
6. Gaede, S. A Scaling Technique for Comparing Interactive System Capacities. CMG XIII Intl. Conf. Proc., Dec. 14-17, 1982, pp62-67.
7. Karels, M. Private communication. May 8, 1984.
8. Pike, R. Private communication. Jan. 25, 1984.

Measuring and Improving the Performance of 4.2BSD

Sam Leffler

Computer Division
Lucasfilm, Ltd.
PO Box 2009
San Rafael, California 94912

Mike Karels
M. Kirk McKusick

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

The 4.2 Berkeley Software Distribution of UNIX[†] for the VAX[‡] has several problems that can severely affect the overall performance of the system. These problems were identified with kernel profiling and system tracing during day to day use. Once potential problem areas had been identified benchmark programs were devised to highlight the bottlenecks. These benchmarks verified that the problems existed and provided a metric against which to validate proposed solutions. This paper examines the performance problems encountered and describes modifications that have been made to the system since the initial distribution. Suggestions for further performance improvements are given.

1. Introduction

The 4.2 Berkeley Software Distribution of UNIX for the VAX has added many new capabilities that were previously unavailable under UNIX. Many new data structures have been added to the system to support these new capabilities. In addition, many of the existing data structures and algorithms have been put to new uses or their old functions placed under increased demand. The effect of these changes is that mechanisms that were well tuned under 4.1BSD no longer provide adequate performance for 4.2BSD. This paper details the work that we have done since the release of 4.2BSD to measure the performance of the system, detect the bottlenecks, and find solutions to remedy them. Most of our tuning has been in the context of the real timesharing systems in our environment. Thus rather than using simulated workloads, we have sought to analyse our tuning efforts under realistic conditions.

2. Observation techniques

There are many tools available for observing the performance of the system. Those that we found most useful are described below.

[†] UNIX is a trademark of Bell Laboratories.

[‡] VAX, MASSBUS, UNIBUS, and DEC are trademarks of Digital Equipment Corporation.

2.1. System maintenance tools

Several standard maintenance programs are invaluable in observing the basic actions of the system. The *vmstat*(1) program is designed to be an aid to monitoring systemwide activity. Together with the *ps*(1) command (as in "ps av"), it can be used to investigate systemwide virtual memory activity. By running *vmstat* when the system is active you can judge the system activity in several dimensions: job distribution, virtual memory load, paging and swapping activity, disk and cpu utilization. Ideally, there should be few blocked (b) jobs, there should be little paging or swapping activity, there should be available bandwidth on the disk devices (most single arms peak out at 30-35 tps in practice), and the user cpu utilization (us) should be high (above 60%).

If the system is busy, then the count of active jobs may be large, and several of these jobs may often be blocked (b). If the virtual memory is active, then the paging demon will be running (sr will be non-zero). It is healthy for the paging demon to free pages when the virtual memory gets active; it is triggered by the amount of free memory dropping below a threshold and increases its pace as free memory goes to zero.

If you run *vmstat* when the system is busy (a "vmstat 1" gives all the numbers computed by the system), you can find imbalances by noting abnormal job distributions. If many processes are blocked (b), then the disk subsystem is overloaded or imbalanced. If you have several non-dma devices or open teletype lines that are "ringing", or user programs that are doing high-speed non-buffered input/output, then the system time may go high (60-70% or higher). It is often possible to pin down the cause of high system time by looking to see if there is excessive context switching (cs), interrupt activity (in) or system call activity (sy). Long term measurements on one of our large machines indicate we average about 60 context switches and interrupts per second and about 90 system calls per second.

If the system is heavily loaded, or if you have little memory for your load (1 megabyte is little in our environment), then the system may be forced to swap. This is likely to be accompanied by a noticeable reduction in system performance and pregnant pauses when interactive jobs such as editors swap out.

A second important program is *iostat*(1). *Iostat* iteratively reports the number of characters read and written to terminals, and, for each disk, the number of transfers per second, kilobytes transferred per second, and the milliseconds per average seek. It also gives the percentage of time the system has spent in user mode, in user mode running low priority (niced) processes, in system mode, and idling.

To compute this information, for each disk, seeks and data transfer completions and the number of words transferred are counted; for terminals collectively, the number of input and output characters are counted. Also, every 100 ms, the state of each disk is examined and a tally is made if the disk is active. From these numbers and the transfer rates of the devices it is possible to determine average seek times for each device.

When filesystems are poorly placed on the available disks, figures reported by *iostat* can be used to pinpoint bottlenecks. Under heavy system load, disk traffic should be spread out among the drives with higher traffic expected to the devices where the root, swap, and /tmp filesystems are located. When multiple disk drives are attached to the same controller, the system will attempt to overlap seek operations with I/O transfers. When seeks are performed, *iostat* will indicate non-zero average seek times. Most modern disk drives should exhibit an average seek time of 25-35 ms.

Terminal traffic reported by *iostat* should be heavily output oriented unless terminal lines are being used for data transfer by programs such as *uucp*. Input and output rates are very system specific. Screen editors such as *vi* and *emacs* tend to exhibit output/input ratios of anywhere from 5/1 to 8/1. On one of our largest systems, 88 terminal lines plus 32 pseudo terminals, we observed an average of 180 characters/second input and 450 characters/second output over 4 days of operation.

2.2. Kernel profiling

It is simple to build a 4.2BSD kernel that will automatically collect profiling information as it operates simply by specifying the `-p` option to `config(8)` when configuring a kernel. The program counter sampling can be driven by the system clock, or by an alternate real time clock. The latter is highly recommended as use of the system clock results in statistical anomalies in accounting for the time spent in the kernel clock routine.

Once a profiling system has been booted statistic gathering is handled by *kgmon(8)*. *Kgmon* allows profiling to be started and stopped and the internal state of the profiling buffers to be dumped. *Kgmon* can also be used to reset the state of the internal buffers to allow multiple experiments to be run without rebooting the machine.

The profiling data is processed with *gprof(1)* to obtain information regarding the system's operation. Profiled systems maintain histograms of the kernel program counter, the number of invocations of each routine, and a dynamic call graph of the executing system. The postprocessing propagates the time spent in each routine along the arcs of the call graph. *Gprof* then generates a listing for each routine in the kernel, sorted according to the time it uses including the time of its call graph descendents. Below each routine entry is shown its (direct) call graph children, and how their times are propagated to this routine. A similar display above the routine shows how this routine's time and the time of its descendents is propagated to its (direct) call graph parents.

A profiled system is about 5-10% larger in its text space because of the calls to count the subroutine invocations. When the system executes, the profiling data is stored in a buffer that is 1.2 times the size of the text space. All the information is summarized in memory, it is not necessary to have a trace file being continuously dumped to disk. The overhead for running a profiled system varies; under normal load we see anywhere from 5-25% of the system time spent in the profiling code. Thus the system is noticeably slower than an unprofiled system, yet is not so bad that it cannot be used in a production environment. This is important since it allows us to gather data in a real environment rather than trying to devise synthetic work loads.

2.3. Kernel tracing

The kernel can be configured to trace certain operations by specifying "options TRACE" in the configuration file. This forces the inclusion of code which records the occurrence of events in *trace records* in a circular buffer in kernel memory. Events may be enabled/disabled selectively while the system is operating. Each trace record contains a time stamp (taken from the VAX hardware time of day clock register), an event identifier, and additional information which is interpreted according to the event type. Buffer cache operations, such as initiating a read, include the disk drive, block number, and transfer size in the trace record. Virtual memory operations, such as a pagein completing, include the virtual address and process id in the trace record. The circular buffer is normally configured to hold 256 16-byte trace records.¹

Several user programs were written to sample and interpret the tracing information. One program runs in the background and periodically reads the circular buffer of trace records. The trace information is compressed, in some instances interpreted to generate additional information, and a summary is written to a file. In addition, the sampling program can also record information from other kernel data structures, such as those interpreted by the *vmstat* program. Data written out to a file is further buffered to minimize I/O load.

Once a trace log has been created, programs which compress and interpret the data may be run to generate graphs showing the data and/or relationships between traced events and system load.

¹ The standard trace facilities distributed with 4.2 actually differ slightly from those described here. The time stamp in the distributed system is calculated from the kernel's time of day variable instead of the VAX hardware register, and the buffer cache trace points do not record the transfer size.

The trace package was used mainly to investigate the operation of the file system buffer cache. The sampling program maintained a history of read-ahead blocks and used the trace information to calculate, for example, percentage of read-ahead blocks used.

2.4. Benchmark programs

Benchmark programs were used in two ways. First, a suite of programs was constructed to calculate the cost of certain basic system operations. Operations such as system call overhead and context switching time are critically important in evaluating the overall performance of a system. Due to the drastic changes in the system between 4.1BSD and 4.2BSD, it was important to verify the overhead of these low level operations had not changed appreciably.

The second use of benchmarks was in exercising suspected bottlenecks. When we suspected a specific problem with the system, a small benchmark program was written to repeatedly use the facility. While these benchmarks are not useful as a general tool they can give quick feedback on whether a hypothesized improvement is really having an effect. It is important to realize that the only real assurance that a change has a beneficial effect is through long term measurements of general timesharing. We have numerous examples where a benchmark program suggests vast improvements while the change in the long term system performance is negligible, and conversely examples in which the benchmark program run more slowly, but the long term system performance improves significantly.

3. Results of our observations

When 4.2 was first installed on several large timesharing systems the degradation in performance was significant. Informal measurements indicated 4.2 performed at about 80% of that of 4.1 (based on load averages observed under a normal timesharing load). Many of the initial problems found were due to programs which were not part of 4.1. Using the techniques described in the previous section and standard process profiling several problems were identified. Later work concentrated on the operation of the kernel itself. In this section we discuss the problems uncovered; in the next section we describe the changes made to the system.

3.1. User programs

3.1.1. Mail system

The mail system was one of the first culprits identified as a major contributor to the degradation in system performance. At Lucasfilm the mail system is very heavily used on one machine, a VAX-11/780 with eight megabytes of memory.² Message traffic is usually between users on the same machine and ranges from person-to-person telephone messages to per-organization distribution lists. After conversion to 4.2, it was immediately noticed that mail to distribution lists of 20 or more people caused the system load to jump by anywhere from 3 to 6 points. The number of processes spawned by the *sendmail* program and the messages sent from *sendmail* to the system logging process, *syslog*, generated significant load both from their execution and their interference with basic system operation. The number of context switches and disk transfers often doubled while *sendmail* operated; the system call rate jumped dramatically. System accounting information consistently showed *sendmail* as the top cpu user on the system.

3.1.2. Network servers

The network services provided in 4.2 add new capabilities to the system, but are not without cost. The system uses one daemon process to accept requests for each network service provided. The presence of a number of such daemons increases the numbers of active processes and files, and requires a larger configuration to support the same number of users. The overhead of the routing and status updates can be appreciable, on the order of several percent of the cpu.

² During part of these observations the machine had only four megabytes of memory.

Remote logins and shells incur more overhead than their local equivalents. For example, a remote login utilizes three processes and a pseudo-terminal handler in addition to the local hardware terminal handler. When using a screen editor, sending and echoing a single character involves four processes on two machines. The additional processes, context switching, network traffic, and terminal handler overhead can roughly triple the load presented by one local terminal user.

3.2. System overhead

To measure the costs of various functions in the kernel, a profiling system was run for a 17 hour period on one of our general timesharing machines. While this is not as reproducible as a synthetic workload, it certainly represents a realistic test. This test was run on several occasions over a three month period. Despite the long period of time that elapsed between the test runs the shape of the profiles, as measured by the number of times each system call entry point was called, were remarkably similar.

These profiles turned up several bottlenecks that are discussed in the next section. Several of these were new to 4.2BSD, but most were caused by an overloading of a mechanism that worked acceptably well in previous BSD systems. The general conclusion from our measurements was that the ratio of user to system time had increased from 45% system / 55% user in 4.1BSD to 57% system / 43% user in 4.2BSD.

3.2.1. Micro-operation benchmarks

To compare certain basic system operations between 4.1BSD and 4.2BSD a suite of benchmark programs was constructed and run on a VAX-11/750 with 4.5 megabytes of physical memory and two disks on a MASSBUS controller. Tests were run with the machine operating in single user mode under both 4.1 and 4.2. Paging was localized to the drive where the root file system was located.

The benchmark programs were modeled after the Kashtan benchmarks, [Kashtan80], with identical sources compiled under each system. The programs and their intended purpose are described briefly prior to the presentation of the results. The benchmark scripts were run twice with the results shown as the average of the two runs. The source code for each program and the shell scripts used during the benchmarks are included in Appendix A.

The set of tests shown in Table 1 was concerned with system operations other than paging. The intent of most benchmarks is clear. The result of running *signocsw* is deducted from the *csw* benchmark to calculate the context switch overhead. The *exec* tests use two different jobs to gauge the cost of overlaying a larger program with a smaller one and vice versa. The "null job" and "big job" differ solely in the size of their data segments, 1 kilobyte versus 256 kilobytes. In both cases the text segment of the parent is larger than that of the child.³ All programs were compiled into the default load format which causes the text segment to be demand paged out of the file system and shared between processes.

The results of these tests are shown in Table 2. If the 4.1 results are scaled to reflect their being run on a VAX-11/750, they correspond closely to those indicated in [Joy80].⁴

In studying the times we find the basic system call and context switching overhead have not changed significantly between 4.1 and 4.2. The *signocsw* results reflect the fact that the *signal* interface has changed, resulting in an additional subroutine invocation for each call, not to mention additional complexity in the system's implementation.

The times for the use of pipes are significantly higher under 4.2 due to their implementation on top of the interprocess communication facilities. Under 4.1 pipes were implemented without the complexity of the socket data structures and with simpler code. Further, while not obviously

³ These tests should also have measured the cost of expanding the text segment; unfortunately time did not permit running additional tests.

⁴ We assume that a VAX-11/750 runs at 60% of the speed of a VAX-11/780 (not considering floating point operations).

Test	Description
syscall	perform 100,000 <i>getpid</i> system calls
csw	perform 10,000 context switches using signals
signocsw	send 10,000 signals to yourself
pipeself4	send 10,000 4-byte messages to yourself
pipeself512	send 10,000 512-byte messages to yourself
pipediscard4	send 10,000 4-byte messages to child who discards
pipediscard512	send 10,000 512-byte messages to child who discards
pipeback4	exchange 10,000 4-byte messages with child
pipeback512	exchange 10,000 512-byte messages with child
forks0	fork-exit-wait 1,000 times
forks1k	sbrk(1024), fault page, fork-exit-wait 1,000 times
forks100k	sbrk(102400), fault pages, fork-exit-wait 1,000 times
vforks0	vfork-exit-wait 1,000 times
vforks1k	sbrk(1024), fault page, vfork-exit-wait 1,000 times
vforks100k	sbrk(102400), fault pages, vfork-exit-wait 1,000 times
execs0null	fork-exec "null job"-exit-wait 1,000 times
execs1knull	sbrk(1024), fault page, fork-exec "null job"-exit-wait 1,000 times
execs100knull	sbrk(102400), fault pages, fork-exec "null job"-exit-wait 1,000 times
vexecs0null	vfork-exec "null job"-exit-wait 1,000 times
vexecs1knull	sbrk(1024), fault page, vfork-exec "null job"-exit-wait 1,000 times
vexecs100knull	sbrk(102400), fault pages, vfork-exec "null job"-exit-wait 1,000 times
execs0big	fork-exec "big job"-exit-wait 1,000 times
execs1kbig	sbrk(1024), fault page, fork-exec "big job"-exit-wait 1,000 times
execs100kbig	sbrk(102400), fault pages, fork-exec "big job"-exit-wait 1,000 times
vexecs0big	vfork-exec "big job"-exit-wait 1,000 times
vexecs1kbig	sbrk(1024), fault pages, vfork-exec "big job"-exit-wait 1,000 times
vexecs100kbig	sbrk(102400), fault pages, vfork-exec "big job"-exit-wait 1,000 times

Table 1. Benchmark programs.

a factor here, 4.2 pipes have less system buffer space provided them than 4.1 pipes.

The exec tests shown in Table 2 were performed with 34 bytes of environment information under 4.1 and 40 bytes under 4.2. To figure the cost of passing data through the environment, the *execs0null* and *execs1knull* tests were rerun with 1065 additional bytes of data. The results are shown in Table 3.

Test	Real		User		System	
	4.1	4.2	4.1	4.2	4.1	4.2
<i>execs0null</i>	197.0	229.0	4.1	2.6	167.8	212.3
<i>execs1knull</i>	199.0	230.0	4.2	2.6	170.4	214.9

Table 3. Benchmark results with "large" environment (all times in seconds).

These results indicate passing argument data is significantly higher than under 4.1: 121 ms/byte versus 93 ms/byte. Even using this factor to adjust the basic overhead of an *exec* system call, this facility is more costly under 4.2 than under 4.1.

3.2.2. Path name translation

The single most expensive function performed by the kernel is path name translation. This has been true in almost every UNIX kernel [Mosher80]; we find that our general time sharing systems do about 500,000 name translations per day.

Test	Real		User		System	
	4.1	4.2	4.1	4.2	4.1	4.2
syscall	28.0	29.0	4.5	5.3	23.9	23.7
csw	45.0	44.0	3.5	3.7	19.5	18.7
signocsw	16.5	22.0	1.9	2.3	14.6	20.3
pipeSelf4	21.5	29.0	1.1	1.1	20.1	28.0
pipeSelf512	47.5	59.0	1.2	1.2	46.1	58.3
pipeDiscard4	32.0	42.0	3.2	3.7	15.5	18.8
pipeDiscard512	61.0	76.0	3.1	2.1	29.7	36.4
pipeback4	57.0	75.0	2.9	3.2	25.1	34.2
pipeback512	110.0	138.0	3.1	3.4	52.2	65.7
forks0	37.5	41.0	0.5	0.3	34.5	37.6
forks1k	40.0	43.0	0.4	0.3	36.0	38.8
forks100k	217.5	223.0	0.7	0.6	214.3	218.4
vforks0	34.5	37.0	0.5	0.6	27.3	28.5
vforks1k	35.0	37.0	0.6	0.8	27.2	28.6
vforks100k	35.0	37.0	0.6	0.8	27.6	28.9
execs0null	97.5	92.0	3.8	2.4	68.7	82.5
execs1knull	99.0	100.0	4.1	1.9	70.5	86.8
execs100knull	283.5	278.0	4.8	2.8	251.9	269.3
vexecs0null	100.0	92.0	5.1	2.7	63.7	76.8
vexecs1knull	100.0	91.0	5.2	2.8	63.2	77.1
vexecs100knull	100.0	92.0	5.1	3.0	64.0	77.7
execs0big	129.0	201.0	4.0	3.0	102.6	153.5
execs1kbig	130.0	202.0	3.7	3.0	104.7	155.5
execs100kbig	318.0	385.0	4.8	3.1	286.6	339.1
vexecs0big	128.0	200.0	4.6	3.5	98.5	149.6
vexecs1kbig	125.0	200.0	4.7	3.5	98.9	149.3
vexecs100kbig	126.0	200.0	4.2	3.4	99.5	151.0

Table 2. Benchmark results (all times in seconds).

Name translations became more expensive in 4.2BSD for several reasons. The single most expensive addition was the symbolic link. Symbolic links have the effect of increasing the average number of components in path names to be translated. As an insidious example, consider the system manager that decides to change /tmp to be a symbolic link to /usr/tmp. A name such as /tmp/tmp1234 that previously required two component translations, now requires four component translations plus the cost of reading the contents of the symbolic link.

The new directory format also changes the characteristics of name translation. The more complex format requires more computation to determine where to place new entries in a directory. Conversely the additional information allows the system to only look at active entries when searching, hence searches of directories that had once grown large but currently have few active entries are checked quickly. The new format also stores the length of each name so that costly string comparisons are only done on names that are the same length as the name being sought.

The net effect of the changes is that the average time to translate a path name in 4.2BSD is 24.2 milliseconds, representing 40% of the time processing system calls, which is 19% of the total cycles in the kernel, or 11% of all cycles executed on the machine. The times are shown in Table 4. We have no comparable times for *namei* under 4.1 though they are certain to be significantly less.

part	time	% of kernel
self	14.3 ms/call	11.3%
child	9.9 ms/call	7.9%
total	24.2 ms/call	19.2%

Table 4. Call times for *namei*.

3.2.3. Clock processing

Nearly 25% of the time spent in the kernel is spent in the clock processing routines. These routines are responsible for implementing timeouts, scheduling the processor, maintaining kernel statistics, and tending various hardware operations such as draining the terminal input silos. Only minimal work is done in the hardware clock interrupt routine (at high priority), the rest is performed (at a lower priority) in a software interrupt handler scheduled by the hardware interrupt handler. In the worst case, with a clock rate of 100 Hz and with every hardware interrupt scheduling a software interrupt, the processor must field 200 interrupts per second. The overhead of simply trapping and returning is 3% of the machine cycles, figuring out that there is nothing to do requires an additional 2%.

3.2.4. Terminal multiplexors

The terminal multiplexors supported by 4.2BSD have programmable receiver silos that may be used in two ways. With the silo disabled, each character received causes an interrupt to the processor. Enabling the receiver silo allows the silo to fill before generating an interrupt, allowing multiple characters to be read for each interrupt. At low rates of input, received characters will not be processed for some time unless the silo is emptied periodically. The 4.2BSD kernel uses the input silos of each terminal multiplexor, and empties each silo on each clock interrupt. This allows high input rates without the cost of per-character interrupts while assuring low latency. However, as character input rates on most machines are usually quite low (about 25 characters per second), this can result in excessive overhead. At the current clock rate of 100 Hz, a machine with 5 terminal multiplexors configured makes 500 calls to the receiver interrupt routines per second. In addition, to achieve acceptable input latency for flow control, each clock interrupt must schedule a software interrupt to run the silo draining routines.⁵ This implies that the worst case estimate for clock processing is, in fact, the basic overhead for clock processing.

3.2.5. Process table management

In 4.2 there are numerous places in the kernel where a linear search of the process table is performed:

- in *exit* to locate and wakeup a process's parent;
- in *wait* when searching for ZOMBIE and STOPPED processes;
- in *fork* when allocating a new process table slot and counting the number of processes already created by a user;
- in *newproc*, to verify that a process id assigned to a new process is not currently in use;
- in *kill* and *gsignal* to locate all processes to which a signal should be delivered;
- in *schedcpu* when adjusting the process priorities every second; and
- in *sched* when locating a process to swap out and/or swap in.

These linear searches can incur significant overhead. The rule for calculating the size of the process table is:

$$nproc = 20 + 8 * maxusers$$

⁵ It is not possible to check the input silos at the time of the actual clock interrupt without modifying the terminal line disciplines, as the input queues may not be in a consistent state.

which means a 48 user system will have a 404 slot process table. With the addition of network services in 4.2, as many as a dozen server processes may be maintained simply to await incoming requests. These servers are normally created at boot time which causes them to be allocated slots near the beginning of the process table. This means that process table searches under 4.2 are likely to take significantly longer than under 4.1. System profiling indicates that as much as 20% of the time spent in the kernel on a loaded system (a VAX-11/780) can be spent in *schedcpu* and, on average, 5-10% of the kernel time is spent in *schedcpu*. The other searches of the proc table are similarly affected. This indicates the system can no longer tolerate using linear searches of the process table.

3.2.6. File system buffer cache

The trace facilities described in section 2.3 were used to gather statistics on the performance of the buffer cache. We were interested in measuring the effectiveness of the cache and the read-ahead policies. With the file system block size in 4.2 four to eight times that of a 4.1 file system, we were concerned that large amounts of read-ahead might be performed without being used. Also, we were interested in seeing if the rules used to size the buffer cache at boot time were severely affecting the overall cache operation.

The tracing package was run over a three hour period during a peak mid-afternoon period on a VAX 11/780 with four megabytes of physical memory. This resulted in a buffer cache containing 400 kilobytes of memory spread among 50 to 200 buffers (the actual number of buffers depends on the size mix of disk blocks being read at any given time). The pertinent configuration information is shown in Table 5.

Controller	Drive	Device	File System
DEC MASSBUS	DEC RP06	hp0d	/usr
		hp0b	swap
Emulex SC780	Fujitsu Eagle	hp1a	/usr/spool/news
		hp1b	swap
		hp1e	/usr/src
		hp1d	/u0 (users)
	Fujitsu Eagle	hp2a	/tmp
		hp2b	swap
		hp2d	/u1 (users)
	Fujitsu Eagle	hp3a	/

Table 5. Active file systems during buffer cache tests.

During the test period the load average ranged from 2 to 13 with an average of 5. The system had no idle time, 43% user time, and 57% system time. The system averaged 90 interrupts per second (excluding the system clock interrupts), 220 system calls per second, and 50 context switches per second (40 voluntary, 10 involuntary).

The active virtual memory (the sum of the address space sizes of all jobs that have run in the previous twenty seconds) over the period ranged from 2 to 6 megabytes with an average of 3.5 megabytes. There was no swapping, though the page daemon was inspecting about 25 pages per second.

On average 250 requests to read disk blocks were initiated per second. These include read requests for file blocks made by user programs as well as requests initiated by the system. System reads include requests for indexing information to determine where a file's next data block resides, file system layout maps to allocate new data blocks, and requests for directory contents needed to do path name translations.

On average, an 85% cache hit rate was observed for read requests. Thus only 37 disk reads were initiated per second. In addition, 5 read-ahead requests were made each second filling about 20% of the buffer pool. Despite the policies to rapidly reuse read-ahead buffers that remain

unclaimed, more than 90% of the read-ahead buffers were used.

These measurements indicated that the buffer cache was working effectively. Independent tests have also indicated that the size of the buffer cache may be reduced significantly on memory-poor system without severe effects; we have not, as yet, tested this conjecture [Shannon83].

3.2.7. Network subsystem

The overhead associated with the network facilities found in 4.2 is often difficult to gauge without profiling the system. This is because most input processing is performed in modules scheduled with software interrupts. As a result, the system time spent performing protocol processing is rarely attributed to the processes which actually receive the data. Since the protocols supported by 4.2 can involve significant overhead this was a serious concern. Results from a profiled kernel indicate an average of 5% of the system time is spent performing network input and timer processing in our environment (a 3Mb/s Ethernet with most traffic using TCP). This figure can vary significantly depending on the network hardware used, the average message size, and whether packet reassembly is required at the network layer. On one machine we profiled over a 17 hour period (our gateway to the ARPANET) 206,000 input messages accounted for 2.4% of the system time, while another 0.6% of the system time was spent performing protocol timer processing. This machine was configured with an ACC LH/DH IMP interface and a DMA 3Mb/s Ethernet controller.

The performance of TCP over slower long-haul networks was degraded substantially by two problems. The first problem was a bug that prevented round-trip timing measurements from being made, thus increasing retransmissions unnecessarily. The second was a problem with the maximum segment size chosen by TCP, which was well-tuned for Ethernet, but which was poorly chosen for the ARPANET, where it causes packet fragmentation. (The maximum segment size was actually negotiated upwards to a value which resulted in excessive fragmentation.)

3.2.8. Virtual memory subsystem

We ran a set of tests intended to exercise the virtual memory system under both 4.1 and 4.2. The tests are described in Table 6. The test programs dynamically allocated a 7.3 Megabyte array (using *sbrk(2)*) then referenced pages in the array either: sequentially, in a purely random fashion, or such that the distance between successive pages accessed was randomly selected from a Gaussian distribution. In the last case, successive runs were made with increasing standard deviations.

Test	Description
seqpage	sequentially touch pages, 10 iterations
seqpage-v	as above, but first make <i>vadvise(2)</i> call
randpage	touch random page 30,000 times
randpage-v	as above, but first make <i>vadvise</i> call
gausspage.1	30,000 Gaussian accesses, standard deviation of 1
gausspage.10	as above, standard deviation of 10
gausspage.30	as above, standard deviation of 30
gausspage.40	as above, standard deviation of 40
gausspage.50	as above, standard deviation of 50
gausspage.60	as above, standard deviation of 60
gausspage.80	as above, standard deviation of 80
gausspage.inf	as above, standard deviation of 10,000

Table 6. Paging benchmark programs.

The results in Table 7 show how the additional memory requirements of 4.2 can generate more work for the paging system. Under 4.1, the system used 0.5 of the 4.5 megabytes of physical memory on the test machine; under 4.2 it used nearly 1 megabyte of physical memory.⁶ This

⁶ The 4.1 system used for testing was actually a 4.1a system configured with networking facilities and code to support remote file access. The 4.2 system also included the remote file access code. Since both systems

resulted in more page faults and, hence, more system time. To establish a common ground on which to compare the paging routines of each system, we check instead the average page fault service times for those test runs which had a statistically significant number of random page faults. These figures, shown in Table 8, indicate no significant difference between the two systems in the area of page fault servicing. We currently have no explanation for the results of the sequential paging tests.

Test	Real		User		System		Page Faults	
	4.1	4.2	4.1	4.2	4.1	4.2	4.1	4.2
seqpage	959	1126	16.7	12.8	197.0	213.0	17132	17113
seqpage-v	579	812	3.8	5.3	216.0	237.7	8394	8351
randpage	571	569	6.7	7.6	64.0	77.2	8085	9776
randpage-v	572	562	6.1	7.3	62.2	77.5	8126	9852
gausspage.1	25	24	23.6	23.8	0.8	0.8	8	8
gausspage.10	26	26	22.7	23.0	3.2	3.6	2	2
gausspage.30	34	33	25.0	24.8	8.6	8.9	2	2
gausspage.40	42	81	23.9	25.0	11.5	13.6	3	260
gausspage.50	113	175	24.2	26.2	19.6	26.3	784	1851
gausspage.60	191	234	27.6	26.7	27.4	36.0	2067	3177
gausspage.80	312	329	28.0	27.9	41.5	52.0	3933	5105
gausspage.inf	619	621	82.9	85.6	68.3	81.5	8046	9650

Table 7. Paging benchmark results (all times in seconds).

Test	Page Faults		PFST	
	4.1	4.2	4.1	4.2
randpage	8085	9776	791	789
randpage-v	8126	9852	765	786
gausspage.inf	8046	9650	848	844

Table 8. Page fault service times (all times in microseconds).

4. System Changes

This section outlines the changes made to the system since the 4.2 distribution. The changes reported here were made in response to the problems described in section 3.

4.1. User programs

Several changes were made in the C library which affected many user programs.

4.1.1. Hashed data bases

A new version of the *dbm* (3X) library was created which supported multiple open data base files per process. This was then used to rewrite the access routines for the password and host data bases. These changes had most significant impact on the performance of the mail system, particularly in a large user and/or host environment (e.g. the ARPANET).

4.1.2. Buffering I/O

The new filesystem with its larger block sizes allows better performance, but it is possible to degrade system performance by performing numerous small transfers rather than using

would be larger than similarly configured "vanilla" 4.1 or 4.2 system, we consider our conclusions to still be valid.

appropriately-sized buffers. The standard I/O library automatically determines the optimal buffer size for each file. Some C library routines and commonly-used programs use low-level I/O or their own buffering, however. One such problem was found in the *ttyslot* library function, which read from the *ttys* file one character at a time. This was changed so that reads were buffered. Two other problems were found in the loader and the assembler, both of which used their own buffering schemes. One kilobyte buffers, as opposed to buffers equal in size to the filesystem block size were discovered. Both have been changed to choose their buffer sizes appropriately for the underlying filesystem.

The standard error output has traditionally been unbuffered in order to prevent delay in presenting the output to the user, and to prevent it from being lost if buffers are not flushed. The inordinate expense of sending single-byte packets through the network led us to impose a buffering scheme on the standard error stream. Within a single call to *fprintf*, all output is buffered temporarily. Before the call returns, all output is flushed and the stream is again marked unbuffered. As before, the normal block or line buffering mechanisms can be used instead of the default behavior.

It is possible for programs with good intentions to unintentionally defeat the standard I/O library's choice of I/O buffer size by using the *setbuf* call to assign an output buffer. Due to portability requirements, the default buffer size provided by *setbuf* is 1024 bytes; this can lead, once again, to added overhead. One such program with this problem was *cat*; there are undoubtedly other standard system utilities with similar problems as the system has changed much since they were originally written.

4.1.3. Mail system

The problems indicated in section 3.1.1 prompted significant work on the entire mail system. The first problem identified was a bug in the *syslog* program. The mail delivery program, *sendmail* logs all mail transactions through this process with the 4.2 interprocess communication facilities. *Syslog* then records the information in a log file. Unfortunately, *syslog* was performing a *sync* operation after each message it received, whether it was logged to a file or not. This wreaked havoc on the effectiveness of the buffer cache and explained, to a large extent, why sending mail to large distribution lists generated such a heavy load on the system (one *syslog* message was generated for each message recipient causing almost a continuous sequence of *sync* operations).

The hashed data base files were installed in all mail programs, resulting in a order of magnitude speedup on large distribution lists. The code in */bin/mail* which notifies the *comsat* program when mail has been delivered to a user was changed to cache host table lookups, resulting in a similar speedup on large distribution lists. Next, the file locking facilities provided in 4.2, *flock* (2), were used in place of the old locking mechanism. This yielded another 10% cut in the basic overhead of delivering mail. Finally *sendmail* was compiled without debugging code, reducing the overhead by another 5%.

The resultant system, while much faster than that originally distributed with 4.2, was still too costly to run at Lucasfilm. This forced the *sendmail* program to be replaced with a simpler delivery system, *sm*, which is 2-3 times faster than the revamped *sendmail* [Ostby84]. The speed is gained through:

- smaller code (the work performed by *sendmail* is distributed among many programs),
- no configuration file (everything is compiled in),
- simpler address parsing,
- buffering small mail messages in memory rather than rereading a temporary file⁷,

⁷ This can result in messages being lost if the system crashes while the message is buffered in memory. Insuring this does not happen is possible with the proper *sendmail* configuration, but is costly since it requires several disk writes per message.

- performing local mail delivery directly, and
- performing fewer forks.

In addition *sm* logs only critical errors.

4.1.4. Network servers

The overhead generated by having one server process always present listening for each service caused a redesign of the basic mechanism by which a server program is created. Rather than having many servers started at boot time, a single server, *ineld* was substituted. This process reads a simple configuration file which specifies the services the system is willing to support and listens for service requests on each service's Internet port. When a client requests service the appropriate server is created and passed a service connection as its standard input. Servers which require the identity of their client may use the *getpeername* system call; likewise *getsockname* may be used to find out a server's local address without consulting data base files. This scheme is very attractive for several reasons:

- it eliminates as many as a dozen processes, easing system overhead and allowing the file and text tables to be made smaller,
- servers need not contain the code required to handle connection queueing, simplifying the programs, and
- installing and/or replacing servers becomes simpler.

With an increased numbers of networks, both local and external to Berkeley, we found that the overhead of the routing process was becoming inordinately high. Several changes were made in the routing daemon to reduce this load. Routes to external networks are no longer exchanged by routers on the internal machines, only a route to a default gateway. This reduces the amount of network traffic and the time required to process routing messages. In addition, the routing daemon was profiled and functions responsible for large amounts of time were optimized. The major changes were a faster hashing scheme, and inline expansions of the ubiquitous byte-swapping functions.

Under certain circumstances, when output was blocked, attempts by the remote login process to send output to the user were rejected by the system, although a prior *select* call had indicated that data could be sent. This resulted in continuous attempts to write the data until the remote user restarted output. This problem was initially avoided in the remote login handler, and the original problem in the kernel has since been corrected.

4.2. Kernel changes

Several changes were made to speed up the bottlenecks discovered in the kernel.

4.2.1. Name cacheing

The system measurements collected indicated the pathname translation routine, *namei*, was clearly worth optimizing. An inspection of *namei* shows that it consists of two nested loops. The outer loop is traversed once per pathname component. The inner loop performs a linear search through a directory looking for a particular pathname component.

Our first idea was to use the fact that many programs step through a directory performing an operation on each entry in turn. This caused us to modify *namei* to cache the directory offset of the last pathname component looked up by a process. The cached offset is then used as the point at which a search in the same directory begins. Changing directories invalidates the cache, as does modifying the directory. For programs which step sequentially through a directory with *N* files, search time decreases from $O(N^2)$ to $O(N)$.

The cost of the cache is about 20 lines of code (about 0.2 kilobytes) and 16 bytes per process, with the cached data stored in a process's *user* vector.

As a quick benchmark to verify the effectiveness of the cache we ran "*ls -l*" on a directory containing 600 files. Before the per-process cache this command used 22.3 seconds of system

time. After adding the cache the program used the same amount of user time, but the system time dropped to 3.3 seconds.

This change prompted our rerunning a profiled system on a machine containing the new *namei*. The results indicated the time in *namei* dropped by only 2.6 ms/call and still accounted for 36% of the system call time, 18% of the kernel, or about 10% of all the machine cycles. This amounted to a drop in system time from 57% to about 55%. The results are shown in Table 9.

part	time	% of kernel
self	11.0 ms/call	9.2%
child	10.6 ms/call	8.9%
total	21.6 ms/call	18.1%

Table 9. Call times for *namei* with per-process cache.

The relatively small performance improvement was caused by a low cache hit ratio. Although the cache was 90% effective when hit, it was only usable on about 25% of the names being translated. An additional reason for the small improvement was that although the amount of time spent in *namei* itself decreased substantially, more time was spent in the routines that it called since each directory had to be accessed twice; once to search from the middle to the end, and once to search from the beginning to the middle.

Most missed names were caused by path name components other than the last. Thus Robert Elz introduced a cache of most recent name translations⁸. This had the effect of short circuiting the outer loop of *namei*. For each path name component, *namei* first looks in its cache of recent translations for the needed name. If it exists, the directory search can be completely eliminated. If the name is not recognized, then the per-process cache may still be useful in reducing the directory search time. The two cacheing schemes complement each other well.

The cost of the name cache is about 200 lines of code (about 1.2 kilobytes) and 44 bytes per cache entry. Depending on the size of the system, about 200 to 1000 entries will normally be configured, using 10-44 kilobytes of physical memory. The name cache is resident in memory at all times.

After adding the system wide name cache we reran "ls -l" on the same directory. The user time remained the same, however the system time rose slightly to 3.7 seconds. This was not surprising as *namei* now had to maintain the cache, but was never able to make any use of it.

Another profiled system was created and measurements were collected over a 17 hour period. These measurements indicated a 6 ms/call decrease in *namei*, with *namei* accounting for only 31% of the system call time, 16% of the time in the kernel, or about 7% of all the machine cycles. System time dropped from 55% to about 49%. The results are shown in Table 10.

part	time	% of kernel
self	9.5 ms/call	9.6%
child	6.1 ms/call	6.1%
total	15.6 ms/call	15.7%

Table 10. Call times for *namei* with both caches.

Statistics on the performance of both caches indicate the large performance improvement is caused by the high hit ratio. On the profiled system a 60% hit rate was observed in the system wide cache. This, coupled with the 25% hit rate in the per-process offset cache yielded an effective cache hit rate of 85%. While the system wide cache reduces both the amount of time in

⁸ The cache is keyed on a name and the inode and device number of the directory that contains it. Associated with each entry is a pointer to the corresponding entry in the inode table.

the routines that *namei* calls as well as *namei* itself (since fewer directories need to be accessed or searched), it is interesting to note that the actual percentage of system time spent in *namei* itself increases even though the actual time per call decreases. This is because less total time is being spent in the kernel, hence a smaller absolute time becomes a larger total percentage.

4.2.2. Auto-siloing terminal input

We observed a low rate of terminal input on most of our systems, which motivated us to re-enable interrupts on a per-character basis. This would allow us to save the high overhead incurred by the system in draining the input silos of the terminal multiplexors at each clock interrupt. Unfortunately, this change would result in huge interrupt loads during periods of heavy input from networks, high-speed devices or malfunctioning terminal connections. We therefore changed the terminal multiplexor handlers to dynamically choose between the use of the silo and the use of per-character interrupts. At low input rates the handler processes characters on an interrupt basis, avoiding the overhead of checking each interface on each clock interrupt. During periods of sustained input, the handler enables the silo and starts a timer to drain input. This timer runs less frequently than the clock interrupts, and is used only when there is a substantial amount of input. The transition from using silos to an interrupt per character is damped to minimize the number of transitions with bursty traffic (such as in network communication). Input characters serve to flush the silo, preventing long latency. By switching between these two modes of operation dynamically, the overhead of checking the silos is incurred only when necessary.

In addition to the savings in the terminal handlers, the clock interrupt routine is no longer required to schedule a software interrupt after each hardware interrupt for the purpose of draining the silos. The software-interrupt level portion of the clock routine is only needed when timers expire or the current user process is collecting an execution profile. Accordingly, the number of interrupts attributable to clock processing is substantially reduced.

4.2.3. Process table management

As noted in section 3.2.5, the linear searches of the process table can result in significant overhead. Consequently, we incorporated changes made by Robert Elz to eliminate all linear searches of the process table. Three separate linked lists are maintained for all: active (allocated) process table entries, inactive (unallocated) process table entries, and for processes in zombie state. These lists eliminate the most expensive process table searches performed by the system. In addition, pointers in the process structure which maintain related processes in a tree structure and which previously had been maintained but not used, were finally used to eliminate the linear searches for parent and sibling processes. These changes were incorporated too late for us to accurately measure the reduction in system overhead.

5. Future work

Many areas for further work still exist. There is still a need to reduce the overhead introduced by the revised system call interfaces for pipes and signals, and for existing facilities such as *exec*. The system wide name cache does not currently support the inclusion of "." and ".."; adding this capability may significantly increase the hit rate. The 100 Hz clock rate needs to be more carefully examined. The initial motivation for this change, to increase the precision of all timing facilities, must be weighed against the basic clock processing overhead. Tom Ferrin has experimented with cutting the clock rate in half with some success [Ferrin84]; it is unclear whether this will result in a significant gain given the changes already made to the clock handling code. Finally, several anomalous test results need to be understood and the late changes to the handling of the process table need to be evaluated.

6. Conclusions

4.2BSD, while functionally superior to 4.1BSD, lacked much of the performance tuning required of a good system. We found that the distributed system spent 10-20% more time in the kernel than 4.1. This added overhead combined with problems with several user programs

severely limited the overall performance of the system in a general timesharing environment.

Changes made to the system since the 4.2 distribution have eliminated most of the added system overhead by replacing old algorithms and/or introducing additional caching schemes. The combined caches added to the name translation process reduce the average cost of translating a pathname to an inode by 35%. These changes reduce the percentage of time spent running in the system by nearly 9%.

The use of silo input on terminal ports only when necessary has allowed the system to avoid a large amount of software interrupt processing. Observations indicate the system is forced to field about 25% fewer interrupts than before.

The kernel changes, combined with many bug fixes, make the system much more responsive in a general timesharing environment. The system now appears capable of supporting loads at least as large as supported under 4.1 while providing all the new interprocess communication, networking, and file system facilities.

Acknowledgements

We would like to thank Robert Elz for sharing his ideas and his code for caching system wide names and searching the process table. We also acknowledge George Goble who dropped many of our changes into his production system and reported back fixes to the disasters that they caused. Eben Ostby did the work on the mail system. The buffer cache read-ahead trace package was based on a program written by Jim Lawson. Ralph Campbell implemented several of the C library changes. The original version of the Internet daemon was written by Bill Joy. In addition, we would like to thank the many other people that contributed ideas, information, and work while the system was undergoing change.

References

- [Ferrin84] Ferrin, Tom, private communication, January 1984.
- [Joy80] Joy, William, "Comments on the performance of UNIX on the VAX", Computer System Research Group, U.C. Berkeley. April 1980.
- [Kashtan80] Kashtan, David L., "UNIX and VMS, Some Performance Comparisons", SRI International. February 1980.
- [Mosher80] Mosher, David, "UNIX Performance, an Introspection", Presented at the Boulder, Colorado Usenix Conference, January 1980. Copies of the paper are available from Computer System Research Group, U.C. Berkeley.
- [Ostby84] Ostby, Eben, "SM: A Small Mailer", Lucasfilm TM. April 25, 1984.
- [Ritchie74] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7. July 1974. pp 365-375
- [Shannon83] Shannon, W., private communication, July 1983

Appendix A – Benchmark sources

Due to the length of this paper the source code for the benchmark programs has not been included. The listings are available, in the form of this appendix, by writing the Computer Systems Research Group at the address given on the front of this document.

The UNIX System *help* Facility

T. W. Butler and L. A. Kennedy

AT&T Bell Laboratories
Summit, New Jersey 07901

ABSTRACT

The new UNIX* system *help* facility provides quick, easy access to information about the UNIX system. It consists of five UNIX commands: *help*, *starter*, *glossary*, *usage*, and *locate*. Each of these commands is the name of a separate module of the facility.

Each module contains a different kind of information:

- *help*: is the top-level menu interface to the entire facility. Through it, all other modules can be entered.
- *starter*: contains information about the local UNIX system, and general information for beginning system users.
- *glossary*: contains definitions of terms and symbols important for the UNIX system.
- *usage*: contains detailed information about specific UNIX system commands.
- *locate*: is a command "thesaurus." It identifies UNIX system commands related to keywords submitted by the user.

The facility is quite flexible, and most information can be accessed in several different ways, depending on the proficiency and preference of the user. All information in the facility can be retrieved by typing "help" at shell command level, and then entering menu choices until the desired information is reached. Users can also use individual modules of the facility by typing the name of the module at shell command level, thereby avoiding the use of some menus. The *glossary*, *locate*, and *usage* commands also accept command-line arguments, so that users can enter all required information on the command line and directly retrieve the information they desire.

The facility also contains a mechanism to collect and store data about its use, and a set of software tools for use by command developers and system administrators who wish to modify the *help* data base.

Much of the data contained in the facility was collected directly from UNIX system users using questionnaires and surveys. The user interface of the facility was designed with special care to be terse enough for experienced users, but simple enough for beginners.

The first release of the UNIX system *help* facility is a first step toward providing a comprehensive, easy-to-use interface between users and all system documentation. Further expansion and refinements of the facility are now under way.

Introduction

The wide proliferation of the UNIX* operating system is a strong indicator of its power and ease of use for programmers and other computer professionals. It is not, however, always the easiest system for beginners to use. The shortcomings of the UNIX system have been widely discussed and need no further elaboration here. One early, and obvious, step we can take toward making the UNIX system easier to learn is to give its users a helpful on-line assistance facility. This

article describes the initial *help* facility we have designed to fill this need.

The first section of the article briefly describes a prototype facility that we built, and the reasons we built it. The next section describes the initial set of requirements adopted for the first version of the facility. This is followed by a discussion of data-gathering to collect information for inclusion in the facility, and, finally, by a discussion of the human interface we designed.

The Prototype

Our first decision was to build a prototype of the facility, and to throw it away at the end of the proposed seven-month development period. While this may seem an odd way to

* UNIX is a trademark of AT&T Bell Laboratories.

begin a development project with short deadlines, the idea did not originate with us (see [1], pp 115-123), and the prototype approach has several distinct advantages. The most important of these are: (1) The prototype could be implemented quickly, (2) the code need not be optimized until we are sure we have designed the system we really want, and (3) the prototype could be tested with real users and revised before it is actually released as a UNIX product. The prototype we designed and implemented was quite small, but it allowed us to:

- evaluate the user interface to the facility,
- evaluate our data collection procedures,
- evaluate the data collected to populate the facility, and
- evaluate the development method.

Requirements and Design

The UNIX system *help* facility is an interactive utility available at shell command level. It allows users to retrieve a variety of information about the UNIX system while they are working on-line, and to retrieve that information easily and conveniently. The facility consists of the top-level interface *help* and four major modules, *starter*, *glossary*, *usage*, and *locate*. The four modules provide the following information, respectively:

1. General information about the local UNIX system,
2. A glossary of technical terms often used in relation to the UNIX system,
3. Information about the usage of specific commands, including examples, and
4. A means of identifying unknown UNIX system commands using keywords related to the desired command functions.

It is easiest to think of the *help* facility as a tree structure, with the top-level *help* menu interface at the root of the tree and the other four modules subordinate to it. Individually, the four modules are all quite different in their structure, and each of them will be described separately.

The *starter* module can be thought of as a sort of formatted bulletin board. It contains general information for users, much of which is

specific to the local system where the facility is installed. In Release 1 of *help*, *starter* includes:

- A list of the UNIX system commands and terms we believe new users should learn first.
- A list of education centers offering UNIX system training.
- A list of important UNIX system documents for beginners.
- A list of on-line teaching aids and tutorials available for the system.
- A set of local environment information, including the system administrator's name and phone number, the name of the system, and the type of processor running the system.

Users retrieve information from these *starter* components by selecting the appropriate category on a menu. The user interface for the facility will be described in greater detail later in the article.

The *glossary* module provides definitions for many common UNIX system terms and symbols. Many of the definitions contain examples to illustrate the concepts being defined, and references to related UNIX system commands. The first screen of this module includes a complete list of the terms and symbols supported. Like *starter*, the *glossary* module is mainly menu-driven.

The *usage* module gives users specific information about individual UNIX system commands. One part of *usage* provides a syntax summary, and a short paragraph describing the action and uses of the command. Much like the *synopsis* section of the *UNIX System User's Manual*, the syntax summary includes a list of allowable options for the command and a command line showing the relative positions of the command name, its options, and its arguments. The two other parts of *usage* include: (1) A list of all options for the command, and an explanation of each, and (2) examples of typical command lines using the command, and an explanation of what each command line does. Users move through these different sections of *usage* by making menu selections.

The final module, *locate*, accepts a set of keywords from the user that are related to the work he or she wants to do, and then identifies a set of UNIX system commands that are likely to be helpful in doing this work. For example, if a user is seeking a command to print the contents of a file, he or she might enter the keywords "file," "print," "contents," and "output," or some subset of this list. The keywords themselves are entirely free-form; users can submit any keyword that they think is related to the work they want to do. The *locate* module keeps a quite large set of keywords on file for each of the UNIX system commands it supports. After accepting the keywords, *locate* checks the keyword list for each command, and returns the names of all commands having a keyword on file that matches any of those entered by the user. After identifying a command, users can directly access the detailed command information in the *usage* module. If they find that the command they selected is not the one they need, they can return directly to the list produced by *locate* and make another selection. As with the other modules, choices are presented and input is solicited using menus.

In addition to these four modules, *help* contains a logging mechanism to collect and store data about its use, and a set of software tools for use by command developers and system administrators who wish to add information about their local system, add data for local commands, or otherwise modify the *help* data base. These additional features are available through a separate command provided with the *help* facility called *helpadm*. Using this command, system administrators can turn on or turn off the logging of usage data on the facility, and both developers and administrators can add, modify, or delete *starter* information, *glossary* definitions, the descriptions, option descriptions, and examples in *usage*, and the keyword information used by *locate* for each command. It should be possible, for example, to use *helpadm* to insert foreign language text and data into the *help* data base.

Data Gathering

Some of the data contained in the facility was obtained through consulting standard reference documents for the system. For

example, this is the way we obtained all the command syntax summaries contained in the *usage* module. Likewise, information about UNIX system training locations, UNIX system beginner documentation, the names of currently-available on-line teaching aids, and the definitions of terms and symbols were obtained in this way. All the remaining information contained in the system was obtained primarily through questionnaires and surveys administered to a controlled mixture of experienced and inexperienced UNIX system users from within AT&T Bell Laboratories.

After a substantial amount of information was collected with the questionnaires and surveys, we collated the results, checked the data for correctness, and edited all the information for consistency. The important difference between our procedure and the usual one in a design effort of this type is that, rather than relying only on the intuitions and judgements of those involved with designing and developing the facility, the intuitions and judgements of a large sample of both experienced and inexperienced users were collected and considered when we decided what to include and what to exclude from the system.

The importance of the data obtained through surveying real UNIX system users cannot be overemphasized. The contents of all of the following were primarily the result of input from UNIX system users:

- The command descriptions in the *usage* module.
- The examples of typical usage for each command in the *usage* module.
- The command keyword list placed on file for use by the *locate* module.
- The specific set of UNIX system commands included in the suggested "minimal" set given in the *starter* module.

Also, all our decisions about precisely what sorts of information to include in the *starter* module were strongly influenced by user input from surveys. This approach is quite different from the usual strategy of relying on designers' intuitions for the first version of a system, and then testing the users' reactions to it later. So far, we have found it to be a pleasant change.

The Human Interface

Designing the user interface for something like a help facility is frustrating. On one hand, the facility must be simple enough for a beginning user, while, on the other hand, the system must be terse enough that expert users won't be irritated by overly verbose guidance and instruction. We have tried to consider both populations of users in the present design, and have produced a flexible, variable interface that should satisfy both user groups. If the facility does favor one group of users over the other, it is probably biased toward helping beginners. This is appropriate, since beginning system users are the ones that presumably most need an on-line help facility. Even so, several features have been added that allow more experienced users to retrieve the information contained in the system more quickly; these features should become evident as the user interface is described.

Basically, the *help* facility is a menu-driven system. The simplest way to enter the facility is to type "help" at shell command level. This yields a menu that allows the user to select either the *starter*, *glossary*, *usage*, or *locate* modules of the facility. Each of these four modules can also be entered directly from shell command level. For example, the *usage* module can be entered by typing either "usage" or "help usage" at shell command level.

Once inside the *starter*, *glossary*, *usage*, or *locate* module, specific information can be obtained. In *starter*, the information is accessed by making a selection from a menu. In *glossary*, entering the name of a term or symbol retrieves its definition. In *usage* and *locate*, the system requests either a specific command name or a set of keywords, respectively. After processing the input, either the requested *starter* information, the definition, the command information, or a list of appropriate UNIX system commands, respectively, is output to the user's terminal. The *glossary*, *usage*, and *locate* modules can be entered directly from shell command level, also, by typing (for *glossary*) "glossary term" or "help glossary term" (where "term" is a UNIX-related term or symbol), (for *usage*) "usage [-d|-e|-o] command_name" or "help usage [-d|-e|-o] command_name", or (for *locate*) "locate keyword1 [keyword2]" or "help locate keyword1 [keyword2]". In the *usage*

module, the option "-d" retrieves the syntax summary and description of the specified command and is the default, the option "-e" retrieves the examples, and the option "-o" retrieves the option descriptions.

Every screen, whether it is primarily a menu screen or not, contains a list of options available to the user at that point. Obviously, these options vary from screen to screen, and there is not enough space here to describe every screen in the system. Nevertheless, we can describe the options in general terms, and provide a reasonably thorough understanding of how the facility works. At every point, the user has the following set of options:

- The user can proceed with entry to a module of the facility, or with some processing within a module.
- The user can always restart the current module.
- The user can always "advance" to the next screen.
- The user can always exit directly to the UNIX shell.
- The user can always return directly to the initial *help* menu, if that is where the facility was entered.
- The user can always escape to the UNIX shell to execute a command, and then return to their current location in the facility.

Aside from the rules given in the last paragraph, the only general principle we used in designing the facility was to make sure that the information it contains is quite densely packed. Quite justifiably, most users strongly object to retrieving screen after screen of menus and intermediate information before reaching the information they need. Certain advantages do accrue for beginners if they are given only a small amount of information in each display, but we believe these advantages to be far outweighed by the irritation that this sort of procedure causes in experienced users, and by the disorientation they nearly everyone experiences after traversing a long series of screens. Conceptually, the *help* facility described here is never more than three layers (or screens) "deep." This results in a quite dense packing of information.

Conclusion

In summary, the *help* facility is an interactive utility available at shell command level that provides a variety of information on the UNIX system. The facility is menu-driven, but allows direct access at two interior levels from the shell, so that experienced users can retrieve information quickly. In addition, the facility's data can be expanded and customized using a set of software tools provided with the facility.

We spent a considerable amount of time designing the flow of control through the facility, and we believe we have achieved our goal: A terse, but easy-to-use, structure and command language.

References

- [1] Brooks, F.P., *The Mythical Man-Month*, Reading, MA: Addison-Wesley, 1975.

Adventures with Typesetter-Independent TROFF

Mark Kahrs[†]
Lee Moore

Department of Computer Science
University of Rochester
Rochester, NY 14627

Abstract: This report discusses experience with the Typesetter independent TROFF. War stories are related regarding the writing of two output filters for an experimental laser printer and a strange phototypesetter. The report concludes with recommendations for the next writer of a device independent output format (as well as for the writers of other output filters for typesetter independent TROFF).

1. Introduction

At the University of Rochester we have access to three "oddball" output devices – two experimental Xerox prototype laser printers and a Compugraphics Editwriter 7700 model II phototypesetter. Therefore, when the availability of Typesetter Independent Troff (sometimes called "device independent TROFF") was announced, we were excited. This report is about what we found out about (1) the new version of TROFF (2) our printers. Hopefully, this report will offer some advice to future writers of ditroff¹ output filters.

1.1. Overview

We will first review the ditroff organization and then we will discuss the implementation for the PRESS output converter for the Xerox laser printers. Later we will discuss the implementation of the output filter for the Compugraphics Editwriter 7700 and then conclude with some general remarks on ditroff.

[†]The work was performed at the University of Rochester. The first author's current address is: AT&T Bell Laboratories, 2C-455, Murray Hill, NJ 07974

¹ We choose to call it *device* independent because it's hard to consider the Xerox printers typesetters!

2. TROFF

We assume that most readers of this paper are familiar with TROFF and the design of ditroff. For those who are left out in the cold, we recommend Akkerhuis' paper [akk83] or Kernighan's original technical report [ker81] for a description of the changes made to TROFF.

2.1. TROFF Design Quirks

Because TROFF was designed for an inexpensive phototypesetter, it makes several assumptions about the device on the other end. For example, it assumes that the device has all the sizes of a given font. In an analog phototypesetter that changes point size by manipulating a lens, this is a quite reasonable assumption. However, in a digital phototypesetter such an assumption is not reasonable. This design bias continues in ditroff.

Another phototypesetter assumption is the action taken after a character is "flushed". In TROFF, the assumption is that the carriage doesn't move, i.e., that a horizontal move must be generated to move over the character. For both of our output devices types this turned out to be inappropriate. This will be discussed further in section 3.5.

Because of the weirdness of the C/A/T typesetter, some special characters have bold and italic versions and some don't. Thus the TROFF model doesn't make it easy to request a bold or italic Ω where as it might be natural to specify a bold $\frac{1}{4}$ character. This problem is basically due to the separation created by troff between special and "normal" characters.

However, this problem is more pervasive. It is not possible to ask for the bold version of the current font because TROFF doesn't really "understand" that a given font is bold or italic or whatever. Nor is it possible to associate sets of special characters with particular fonts. For example, if one is currently in a bold font, one would also want the bold special characters.

TROFF also lacks the ability to describe rotated fonts. This too, is a direct result of the historical dependence on the C/A/T typesetter.

Not all devices have a fixed coordinate system like the one in TROFF. One of our devices has a coordinate system that varies as a function of the current point size.

The ultimate destination of a TROFF document can be any number of output devices and each device can have wildly different font sets. In our case, the two Xerox laser printers have very different font sets at very different resolutions. Unfortunately, TROFF doesn't allow you to describe a device class and then talk about instances of that class.

Although the description file for the typesetter was supposed to be the "source of all knowledge" about the output device, the filter packages (TBL, EQN, PIC) don't look there. All of the device dependent information is hard coded into the C source of all these filters.

2.2. TROFF War Stories

Herein we relate actual events as they happened to your intrepid reporters.

When we looked at some of the early output from the X2600, a few of the characters were overlapping. Furthermore, the same characters would always overlap. Then we noticed that it would only happen to the widest characters: things like "M" and "W". Why is it, we asked ourselves, that some of the widest characters are treated like they have the thinnest widths? Well, it turns out that the maximum width of a character in a compiled font description file is 255 "whatevers" (goobies at unitwidth), which was not documented *anywhere*. Should a character in the input description file be greater than 255, its high order bits are (silently) truncated. For both of our devices we had to specify widths using rather large cardinal numbers. Fortunately, all the widths were less than 512 so we just halved the unitwidth and recomputed the input tables. (The TROFF tables were automatically generated).

A subtle bias in TROFF comes from the fact that the implementors originally had one particular typesetter. Therefore, there isn't a clean mechanism to separate details about the input language from the device configuration. Thus one is forced to treat machines as device-language pairs rather than a language which can be input to many different instantiations. While this problem can arise with a traditional phototypesetter, it became acute with our Xerox

printers. At one level we wanted to treat all Xerox PRESS devices the same because they all accept the same input language; on the other hand, different printers have different fonts sets and must be treated differently at certain points. An annoying place this occurs is in EQN. It has a hard-coded switch on device type which sets the minimum point size of the device. Since each PRESS device can have a different font set and thus a different minimum point size, the correct solution would have been to declare each different printer as a different device. Partly to retain our sanity, we chose not to do this but it is clear what the problem is.

Several special characters must be defined in order for TROFF to work properly (in some cases, work at all!). These characters are listed in Appendix B.

3. Implementation 1: Press

This section will now discuss the actual printers (the X2600 and Warlord) and how the TROFF converter was written. We will also discuss a few problems that we met and conquered along the way. We will first describe the particulars of our output devices.

3.1. The X2600

The X2600 is a one of a kind prototype that was donated to the University of Rochester by the Xerox Webster Research Center. The print engine is a converted 2600 copier (a small desk-top model) that was modified to include a galvanometer and a small Helium-Neon laser. The resolution of the device is 240 spots/inch both vertically and horizontally. Thus each page has $(8.5)(11)(240)(240) = 5,385,600$ points per page.

The printer is driven by a Xerox Alto [tha81] (not sold in stores) that sends control and video signals to the print engine. The Alto came with a set of programs to receive PRESS formatted files from the Ethernet² and print them. To print a file, the Alto must convert each page to a bitmap, start the 2600 rolling and then shove the bitmap out the door *in real time*. Unfortunately, because of memory constraints in the Alto, this bitmap must be constructed on the (slow) Alto disk. Each page of text typically takes 30 seconds to format and print, although graphics can make it can take *much* longer.³

² Ethernet is a trademark of Xerox Corporation

³ Rumor has it that a certain piece of sheet music took 20 minutes to format!

3.2. The Warlord

The Warlord is another one of a kind prototype donated to the University by Xerox. The printing engine is an XP-12, the same device found in a model 2700 printer, the DEC LN01 and the QMS Lasergrafix 1200 (tm). The XP-12 is connected to a custom interface called a "DoLI" that converts the output of the Alto OrBit cards (see [tha81] for a description of the OrBit) to the video format of the XP-12. Note that the OrBit is totally different than the interface used by the 2600; in particular, the formatting is done in the OrBit memory *on the fly* which means that certain pictures (like music!) can be too complex for the printer to print. On the other hand, it is faster for most applications. Note also that the resolution of the XP-12 can be jacked up to 384 spots to the inch by hacking the copier electronics.

The next section describes enough of the PRESS format to give the flavor of the conversion process.

3.3. PRESS

The PRESS format⁴ was designed to be a "lingua franca" of printers inside of Xerox. While not used in products, it was remarkably versatile and it has been used by many processors, programs and printers.

PRESS files basically describe a series of pages to be printed. Each page may have text and/or graphics that can be positioned anywhere on the page. Graphics may be either rectangles (which are supported by virtually every PRESS printer) or they may be defined by outlines (which are not supported on many printers). To obtain maximum generality, we tried to stick with text and rectangles as best we could.

Coordinates in PRESS files are given in *micas* (there are 2540 per inch). The alert reader will realize that the input language has a resolution which is about an order of magnitude greater than the true resolution of either of our PRESS printers. The origin lies in the lower left hand corner of the page with X values increasing to the right and Y values increasing in the upward direction. This is partially in opposition to TROFF where Y units increase in the downward direction.

TROFF has a notion of the current position on the page. The position may be set at any coordinate position. The X and Y coordinates are set separately. When a character is displayed, the current X (and Y) position is always updated. As mentioned before this

does *not* match TROFF's typesetting model. PRESS also has a command to show either an individual character or a string of characters, starting at the current position. This feature will become important later when we discuss the translation of troff intermediate "code" into PRESS.

3.4. Constructing the Width Tables

Like most typesetters, the actual representation of the fonts is stored at the printing device (in this case, the disk of the Alto). Application programs need only to know the name of the fonts on the printer and the width of each character in each font. This is done by means of a binary file called "fonts.widths". It contains a dictionary of fonts with relative pointers to the specific width tables. The widths are of two types: those specific to a particular family/face/rotation/size and those that are scaled to represent all sizes in a particular family/face/rotation. It turned out that all the popular fonts were stored in scaled format. When scaled, the size is stored in units of thousandths of a point. It is converted to micas with the following formula

$$Width_{micas} = \frac{1}{1000} \frac{desiredPointSize}{1} \frac{scaledWidth}{1}$$

$$\frac{2540micas}{inch} \frac{inch}{72points}$$

Clearly, it was desirable to automatically create the font description files for TROFF rather than building them by hand. The following catalogs some of our experience. Initially we had some problems because the fonts.widths file had many more fonts in it than our printer actually had. In addition, most of the fonts that we did have were declared to have more characters than really existed. We can only assume that our fonts.widths file was actually one that belonged to another printer and that it only worked for us because it described printer fonts that were derived from the same spline representation that ours were. As a side note, most of the major fonts are compatible across all the experimental laser printers despite differences in actual resolution. PRESS files made at Xerox as well as those files from other Xerox grant universities tend to print as they should.

Since most of the fonts in fonts.widths were stored in scaled format, we couldn't automatically determine which point sizes were really available on our printer. To generate a correct device description file (DESC) file, we had to augment this file with additional information.

⁴ The (gentle) reader should not confuse PRESS with Interpress, the recently announced Xerox printing standard.

As mentioned before, all the fonts aren't available in all sizes (not even within a family.) Unfortunately, TROFF has a lens-and-mask model of a typesetter. In this model, the point sizes are determined by the position of the lens and thus the size is font independent. In ditroff, the list of sizes is given in the description file (DESC) and thus is assumed to be uniform across all fonts. We decided to take the union of all font sizes and put that in the description file and to pass the true sizes to the output filter. The output filter compares the point size asked for with the available point sizes and chooses the greatest size less than or equal to the asked-for size.

Special characters also represented a problem because they had to be mapped from a position in a font to a TROFF name. Fortunately, there were only two fonts with special characters: hippo (greek) and math. To treat this problem uniformly, description files were created that specified the mapping of characters in those fonts to the TROFF character names. Since all fonts in a particular family had the same mapping, the following algorithm was used: first a directory lookup is done on the filename (concat FamilyName ".map"), if that isn't found then the file "default.map" is used. The file "default.map" is basically the ASCII character set. Using the map files also allows us to exclude those characters that are mentioned in the fonts.widths file but don't exist on our printer. The ascender and descender information is also found in the .map files.

Lastly, the conversion program (called "digest") always creates a file describing the pseudo-font "S". This font doesn't really exist on the printer but contains those symbols that the post-processor will create using the PRESS drawing commands. The bracket building and ruling characters fall into this category. (See Appendix B for the description of these characters).

The "digest" program accepts the fonts.widths file and the .map files for the appropriate device as input. It produces all the font files and the DESC file. In addition, it produces two text files that the output filter (called "dpress") uses to decide what fonts and what sizes are actually on the output device. One important side effect of "digest" is that it preserves the font order of the device description input file in the DESC output file. This is needed because one wants to insure that the Times Roman font (or a reasonable facsimile) is mounted on positions 1, 2 and 3. Naturally, this is for compatibility reasons with macro packages that address fonts by number instead of by name.

Because the DESC file does not describe the output device in enough detail for our application, the "digest" program creates two auxiliary description files. Since the internal name of a font must be represented as a (short) character array, it was not possible to represent the complete family name/face/stress/rotation for an arbitrary font. Also comparisons for equality are expensive in this full n-tuple representation. Instead we chose to internally name fonts by small integers and then lookup the full name at output time. In addition, by having the number of fonts stored in a file, it is an easy matter to allocate the various arrays at run-time rather than using a compiled in "maximum" constant. Another problem was that the output filter needed to know which fonts really were on the printer and which fonts were not. This information is kept in a separate file so that the PRESS subroutine package could be more cleanly separated from any influence of TROFF.

The map description files have the following syntax:

```
{normal | special}
{{octal value} {kern} {TROFF name}*\\n}*

```

An example file is below:

```
#
#   Default mapping of characters
#
normal
#
#       1 = ascender
#       2 = down
#       3 = both
#
# shouldn't define a space (040)
#
#char (a|de)scender      alias
041   1      !
042   1      "
043   1      #
044   1      $
045   1      %
046   1      &
047   1      '
050   1      (
051   1      )
052   1      *
053   1      +
054   2      ,
# almost required to have a "hy"
055   1      - hy
056   0      .
057   1      /

```



```
060 1 0
061 1 1
062 1 2
063 1 3
```

The printer description file has the following syntax:

```
device {device name}
{{family}} {face} {TROFF name} {point Size}*\\n}*
```

An example file is below:

```
device x2600

# Timesroman always goes first

timesroman mr R 6 7 8 10 12 18
timesroman mi I 6 7 8 10 12
timesroman br B 6 7 8 10 12 18
timesroman bi X 6 7 8 10 12

elite mr ER 10

gacha mr GR 8 10

helvetica mr HR 6 7 8 10 12 18
helvetica mi HI 6 7 8 10 12
helvetica br HB 6 7 8 10 12 18
helvetica bi HX 6 7 8 10 12

# start of "special" fonts

hippo mr HP 8 10
math mr MA 6 8 10
```

3.5. Translating the ditroff Output

The output from ditroff is given directly to the "dpress" output filter. Fortunately, we had access to a set of routines (originally coded in SAIL by Ivor Durham and converted to C by Tom Rodeheffer) that created valid PRESS primitives. We just interfaced this library with the program and we almost had it right. Almost.

First, we were faced with the problem of TROFF to PRESS coordinate systems. Eventually, we settled on TROFF coordinate system so that the coordinate system would remain compatible with the graphic (line, circle and ellipse) drawing routines supplied with the ditroff release. Conversion to PRESS coordinates occurs right before calls to the PRESS package.

Second, we wanted to generate compact TROFF files. In general, TROFF will send a positioning command before every character. The obvious implementation of this is to issue a set-position command for each character so that it will be placed exactly. This

was the approach taken by "dcat", an existing program that converted the C/A/T output of the old TROFF. Sadly, this created files that were four or more times longer than then the input text. They were bulky to store and bulky to ship around. In addition, they were difficult to print since they often exceeded the spooling space of the disk on the printer Alto. The solution is easy to see because TROFF doesn't do letter spacing. That is to say, it won't insert space between letters, only between words. We could achieve great compression by using the "show character string" command for each word but this meant that we had to figure out there the word breaks are.

Our first attack at this used the "w" command to delimit words. A character buffer is filled by the characters using the "c" commands. The "w" command empties the character buffer and the following "h" command was used to generate the appropriate "set x" command.

Then we discovered an interaction with tabs. Tabs are characters, not white space. Therefore, TROFF doesn't generate a "w" command after a tab. Without this, it's impossible to get TBL right, much less hanging paragraphs. How did we get around it? The answer is quite simple. Recall that each character generates a "c" command followed by a "h" command. The "h" command moves over the width of the character plus any additional move (such as space between words). Now, if the horizontal move is the same width as the character previously printed, then this width must be redundant and therefore no move is generated. However, if the width isn't equal to the last width, then a "set x" command in PRESS is generated.

The "special characters" of TROFF presented their own problems. Many of them were available on existing fonts but many, notably the bracket building functions, were not. Xerox did provide us with the tools to create new fonts. Unfortunately, on the X2600, it is rather laborious to test new fonts because of the way fonts are stored.⁵ Also locally generated PRESS files which asked for these locally created fonts would not print well at sites other than our own. Our solution was to use C code to emit drawing commands for those characters that could be drawn and to ignore characters like the Bell System logo and the pointing hands. Unfortunately, the design of these characters are *not* specified in the troff document. Likewise, the design of the "baseline rule" and the "underline rule"

⁵Our new press printer (a Warlord) does allow a client to send private fonts along with a PRESS file. This feature greatly facilitates

are left undefined. So that other ditroff interface writers will not lose their sanity, we are including the definition of the bracket building characters in Appendix B. Note that most of the drawing code was stolen (shamelessly) from the C/A/T filter written at CMU.

Hand coding was also present in EQN, TBL and PIC. Both programs required changes to their internals in order to run. At a minimum, the resolution of the output device has to be "wired in" to both programs. They all ignore the already existing description file! In the case of EQN, the problem is exacerbated by the fact that EQN *expects* the typesetter to have a nearly infinite variety of point sizes. If one's output device is somewhat limited, things tend to look crowded because TROFF will substitute the next larger point size for sizes it doesn't have. One strategy we tried for dealing with this problem was to define all possible point sizes and then let the output filter substitute the appropriate size. For equations, it made the output look spacious rather than crowded. Unfortunately, this had bad side effects if you selected the wrong size in normal text and so it was later dropped.

4. Implementation 2: Editwriter 7700

The Compugraphic Editwriter 7700 (model II) is a rather conventional phototypesetter that is aimed at low volume print shops. Its light path begins with a drum with four "slots" for mountable font film strips. The next stage is a lens system after the font strips that can magnify a font up to 36 points. The Editwriter has a RS-232 interface called the Intelligent Communication Interface or "ICI". The ICI interprets character strings that come from the modem into pseudo keystrokes of the input typewriter. This interpreter is actually just a finite state machine, but it permits the user of the ICI to "pretend" that a typesetter is "at their fingertips" by designing a simple language for input. The important thing to note here is that we are trying to force TROFF to generate output for what is a human, not a computer, interface.

4.1. The Phototypesetter

The phototypesetter has a number of characteristics that effect the output converter. To begin with, the typesetter assumes that flashing a character advances the x position (like the PRESS device). Second, the position of words on the page is not done by explicit control of the coordinates (i.e., there are not commands like "set x" and "set y"). Rather, the phototypesetter assumes that the commands are of a very high level, like "center this". The motion

commands are actually spacing commands; they insert space between characters. A space can either be an em, en or "thin" (3/54 of an em) space. This limited notion of spacing conflicts *severely* with the way ditroff looks at a typesetter. While such an interface is clearly designed with humans in mind, it makes the output filter a kludge.

4.2. The Editwriter Interface

First off, we wanted to create a reasonably compact output language for the ICI. Secondly, we wanted to be able to read it. While it wasn't possible to achieve the first goal to the degree that we would have liked, the second was readily obtainable. The input language to the ICI is controlled by a finite-state automata that is sent before the actual text. It controls the mapping from multi-code input sequences to internal command codes. We designed a language with an escape character ("@") that is followed by one, two, or three characters (depending on frequency of use) that specify what character code the sequence represented. The horizontal spacing commands are kept to one letter for the most part for the obvious space saving reason. We have had no problems with this part of the output filter.

4.3. Constructing the Width Tables

After paying a small fee, Compugraphics will supply you with a width table for a given font. Unfortunately, they fail to tell you what units the widths are specified in. Normally, one might think this would be an easy question to answer. Not so. It took at *least* two hours on the telephone to come track down the answer. It turns out that the font widths are given in "units"⁶. These "units" turn out to be the number of 16ths of an inch when the glyph is photographed at 250 points. For example, a character with 48 "units" of width is 3 inches wide (at 250 points). Naturally, these widths must be reduced by the current point size.

4.4. Translating the ditroff Output

The output of TROFF is fed directly to the 7700 output filter. The hardest problem is the generation of the spacing commands. The 7700 doesn't have any absolute motions – all are relative. Furthermore, the distance that a horizontal move command travels is a function of the point size. For that reason, there isn't an absolute coordinate system which means another

debugging and tuning fonts.

⁶ This use of units is *not* to be confused with 18 units per em.

mismatch with TROFF. In the DESC file we defined the resolution to be a close approximation to the coordinate system at a font size of 1 point. Motions are fixed up later in the output filter. As in the PRESS filter, the 7700 filter compresses its output by recognizing words, however it handles motions slightly differently. The translation strategy for this device becomes as follows: The distance of any horizontal move is compared with the width of the last character. If the width is different, then the output filter generates some number of spacing commands (em space, en spaces and so forth) until no more spaces can be fit. The left over "slop" is carried onto the next horizontal move. The "slop" is reset at the beginning of each line. What matters, of course, is the leftover slop at the *end* of a line. Fortunately, statistics taken show that the typical average slop is around 25/4000's (0.006) of an inch. In fact, this document had a average slop of 0.003 (12/4000) of an inch.

Like horizontal positioning, vertical positioning must be approximated. In particular, the vertical positioning commands deal in points. Unfortunately, TROFF insists in dealing with goobies (whatevers per inch). And it just so happens that the resolution we use for the printer does not divide nicely into whatevers per inch. Therefore, the description file for the typesetter uses an approximation for the vertical move. Naturally, this generates a bit of slop, but we chose to ignore it on the theory that the eye of the reader is less sensitive to vertical positioning. Vertical positioning on the Editwriter is generally done after every carriage return. The amount of spacing is set by a "line spacing" command. The strategy for the output filter is very simple: If the spacing being asked for is the same as the last spacing, then just emit a carriage return. If the spacing is different, then emit the command to change the spacing and *then* emit the carriage return.

In a simple typesetter like the Editwriter, certain TROFF commands must be ignored. In particular, the graphic commands and rules are useless on the Editwriter.

Another problem with the Editwriter is not immediately obvious. Although it's possible to fool the typesetter into getting the spacing correct, it also takes a fair amount of space on the floppy disk to do so! In fact, the sectors on the disk rapidly fill up.

The Editwriter 7700 has more than its share of problems. Besides the problems of interfacing to a high level typesetter, figuring out the width tables and designing the interface language, the machine isn't even connected directly to our CPU. Instead, it's

necessary to go through an intermediate, thereby risking various problems along the way. Fortunately, this has not been a major problem, just a pain in the gluteus maximus.

The Editwriter also forced us to discover another problem with TROFF. The symptom of this failure is the message "font # too big for position #", where the #s are numbers. The solution to this problem is to make the binary font width files the same size. This can either be done by hacking the makedev program, by hacking the fonts, or increasing the maximum size of the troff internal buffer. We chose to do the last alternative.

As mentioned earlier, the Editwriter has a hard sector floppy disk where it stores the incoming data. Each sector is considered to be a file by the near-derthal file system. During computer input the Editwriter will start a new file when the current sector (file) is full. Unfortunately, during output (actual typesetting), all state is lost between files. This means that state information must be reset at the front of each file. The output filter attempts to compensate for this by directing the Editwriter to begin a new file near sector boundaries and then emitting the state information.

5. Conclusion

5.1. Our Subjective Feelings

The first question is ask is: was it worth it? Well, yes and no. Yes, the X2600 was worth all our efforts. It's hard to tell about the 7700.

5.1.1. X2600

The X2600 prototype was everything that a prototype could be. Unreliable. Flakey. Slow. But... it actually works (sometimes); and when it does work well, it works. At this date, the X2600 has been replaced by the Warlord printer described earlier connected to yet another Alto⁷ running a different PRESS formatter. This improved the output quality, font catalog and reliability immensely (as predicted).

5.1.2. Editwriter 7700

It was clear to us very early on that the 7700 had the wrong interface for TROFF. Obviously, we would have liked to plug directly into the output device rather than through the Editwriter. Still, we persevered because we thought we could at least get a "hack" solution. Our solution required to us to

generate huge files because of the lack of concise and precise position commands. We also realized early on that this would present a space conflict on the Editwriter's floppy. We made some effort to discover the maximum size of a file on the floppy. Our local people considered themselves to be artists and only had vague ideas about this. Reading the manuals didn't help, either. Although the average floppy file will hold a legal size page of normal text, we found that it would only hold a half page from our post-processor. This considerably diminishes its usefulness.

Our life would have been made considerably easier if TROFF had a more flexible view of the output device. It would be nice if TROFF had a way to describe the level of the typesetter. Part of the problem is that troff has a *very* low level view of possible typesetters. This, of course, is history, since ditroff is just a hacked-up version of the C/A/T typesetter that featured these low level commands.

5.2. Future Recommendations

The output format of TROFF works extremely well in our opinion. It is well designed and easy to interpret. The fact that it is an ASCII file (and readable) makes it relatively easy to debug.

The hand modifications of the input filters demonstrate that something was lacking in the typesetter description file. On one hand, these were mostly hacks to get the output as beautiful as possible, and therefore can be considered "fine tuning". However, the question remains about how to specify such parameters.

Perhaps the greatest area for improvement is the description of both the fonts and the typesetter. In particular, the font descriptions should have the sizes bound to them and not the typesetter. Furthermore, since faces are standard in fonts, there should be a way to relate fonts to one another (this font is the bold version of ...). There should be some way to describe what the primitive commands of the typesetter are and how they react (a semantics of typesetting?). Also there should be a class and subclass specification language that can define a device class and then specialize that class for specific instances. And finally, there should be some way to describe all of the measurement units of a typesetter.

6. Acknowledgements

Clearly, we wouldn't have even have started on this project if it hadn't been for Brian Kernighan's

labors. We thank him for that and for helping us with the "\|" problem. We only hope that he doesn't take our criticism of TROFF personally.

We owe a large debt to the Xerox Webster Research Center for providing us with the X2600 and later the XP-12 based Warlord. It helped entice people (and addict them) to the glories of typesetting.

Shirley Kelly of Compugraphics withstood my (MWK) questioning for several hours. Her assistance in figuring out what "units" *really* meant were crucial to our "success" with the 7700.

Finally, we are indebted to the folks who have written previous PRESS software. This includes (at CMU) James Gosling (for "cz"), Mike Accetta, Thomas Rodeheffer and Ivor Durham and (at Stanford) Bill Nowicki.

7. Bibliography

[akk83]

J. Akkerhuis, Typesetting and TROFF, *Proceedings of the European Unix Systems User Group Autumn Meeting*, Dublin, Ireland, Sep. 1983, 29-41.

[ker81]

B. W. Kernighan, A Typesetter independent TROFF, CSTR 97, Bell Laboratories Report, 1981.

[tha81]

C. W. Thacker, E. M. McCreight, B. W. Lampson, B. W. Sproull and D. R. Boggs, Alto: A Personal Computer, in *Computing Structures: Principles and Examples (2nd. Ed)*, D. Sieworek, C. G. Bell and A. Newell (ed.), McGraw-Hill, New York, 1981, 549-582.

⁷ "Old Altos never die, they just become servers"

APPENDIX A

Handy characters to define

The following table contains a list of characters that should (and in some cases *must*) be defined for ditroff to work properly. This is the voice of experience.

\(hy		Character to be used when troff hyphenates words. If you don't define it, troff will break the word without any indication. (i.e. ditroff won't substitute an ascii "--")
\(em	—	¾ em dash. Used by -me macros
\ \↑	(1/6 em space) (1/12 em space)	Heavily used in EQN. Not defining it will lead to unpredictable gobbling of succeeding characters. Define to have the (reserved) character code 0. They will be eaten by the post-processor.
\(eq \(mi \(pl	= - +	These are the equation version of the regular ascii characters =, -, and +. They are invariant w.r.t. to the current font (i.e. one never gets an italics equal sign.) Used heavily by EQN.
\(ru	—	Used for building up horizontal lines. Two adjacent copies of this character shouldn't have any white space between them. Not defining this character will cause a floating point exception (!!!). Used in TBL, EQN and lots of other places.
\(br		Really a vertical rule that is displaced enough w.r.t. to the current base line so that it mates with the left size of a \(ru to form a seamless corner. Its height at any one time is the current point size. Thus unbroken vertical lines can easily be formed. Used by TBL.
\(sr	✓	Obviously needed for EQN. Must have a width exactly the length of the character (no white space on the right).

APPENDIX B

Special character table

```
#define RESOLUTION      2540 /* micras/inch */
#define EM              (RESOLUTION/PointsPerInch)
#define RULELENGTH      20 /* 20 micras/point of magnification */
#define RULEHEIGHT      EM
#define ULLENGTH        20 /* 20 micras/point of magnification */
/* WidthInFontFile/UnitWidth = 20 (exactly) */
```

These routines were stolen from the CMU dcat program. We hope they don't mind, but we like them (the routines). The routines Move and Rect have their arguments multiplied by the current point size. All of the drawing is thus done as if the current point size is one (1) and then are scaled up before output. The special character drawing routines use the following helper functions:

DrawElbow(xdir,ydir)	Rect(8*xdir, 2*ydir); Move(4*xdir, 2*ydir); Rect(4*xdir, 34*ydir); Move(0*xdir, 34*ydir);
DrawCurl(xdir,ydir,reach)	Rect(4*xdir, 1*ydir); Move(2*xdir, 1*ydir); Rect(3*xdir, 1*ydir); Move(1*xdir, 1*ydir); Rect(3*xdir, 1*ydir); Move(1*xdir, 1*ydir); Rect(3*xdir, 3*ydir); Move(0*xdir, 3*ydir); Rect(4*xdir, (reach-6)*ydir); Move(0*xdir, (reach-6)*ydir);

The following routines generate a box, a bold box, a horizontal rule, a vertical rule and finally, an underline.

square		$\text{length} = \frac{5}{8}em$ $\text{thickness} = \frac{1}{15}em$ Rect(length,thickness); Rect(thickness,length); Move(0, length – thickness); Rect(length,thickness); Move(–(length – thickness), –(length – thickness)); Rect(thickness,length);
rule	–	Rect(RULELENGTH, $\frac{1}{18}em$);
vertical rule		Move(0, $-\frac{5}{36}em$); Rect(RULEHEIGHT, $\frac{1}{18}em$); Move(0, $\frac{5}{36}em$);
underline	–	Move(0, $-\frac{5}{36}em$); Rect(ULLENGTH, $\frac{1}{18}em$); Move(0, $\frac{5}{36}em$);
root en extender		/* the width was chosen to look right with our fonts */ Move(0, em); Rect(RULELENGTH, $\frac{1}{36}em$); Move(0, –em);
$\frac{3}{4}em$ dash	–	Move(0, $\frac{2}{9}em$); Rect($\frac{3}{4}em$, $\frac{1}{18}em$); Move(0, $-\frac{2}{9}em$);

These are the bracket building characters. They are listed on page 33 of the TROFF User's Manual.

<code>\lt</code>	<code> </code>	Move(12-4,36-8); DrawCurl(-1,-1,36);
<code>\lb</code>	<code> </code>	Move(12-4,0-8); DrawCurl(-1,1,36);
<code>\rt</code>	<code> </code>	Move(0-4,36-8); DrawCurl(1,-1,36);
<code>\rb</code>	<code> </code>	Move(0-4,0-8); DrawCurl(1,1,36);
<code>\lk</code>	<code> </code>	Move(0-4,18-8); DrawCurl(1,-1,18); Move(-4,18); DrawCurl(1,1,18);
<code>\rk</code>	<code> </code>	Move(12-4,18-8); DrawCurl(-1,-1,18); Move(4,18); DrawCurl(-1,1,18);
<code>\bv</code>	<code> </code>	Move(4-4,0-8); Rect(4,36);
<code>\lf</code>	<code> </code>	Move(12-4,0-8); DrawElbow(-1,1);
<code>\rf</code>	<code> </code>	Move(0-4,0-8); DrawElbow(1,1);
<code>\lc</code>	<code> </code>	Move(12-4,36-8); DrawElbow(-1,-1);
<code>\rc</code>	<code> </code>	Move(0-4,36-8); DrawElbow(1,-1);

Note that `\bv` (bold vertical) is not the same as `\br` (box vertical rule). Note also that `\rt`, `\rb`, `\lk`, `\rf` and `\rc` all have a slight left kern. This makes the text which is to the left of them slightly squeezed. A more aesthetic solution would have been to define another `\bv` that was slightly offset. Unfortunately, this would also mean changing pre-processor(s).

The Readers Workbench - A System for
Computer Assisted Reading

Evan L. Ivie
234 TMCB
Computer Science Department
Brigham Young University
Provo, Utah 84602
(801) 378-7655

Abstract:

A rapidly increasing body of electronic text is now becoming available from such sources as computer-driven word photocomposers, word processors, and personal computers. This paper describes a set of tools that assist in the examination and assimilation of unstructured electronic text.

In a vein similar to the Programmers Workbench, the Writers Workbench, and the Documenters Workbench from Bell Labs, the Readers Workbench provides utilities that allow one to perform on-line reading more effectively. Examples of Readers Workbench tools include dictionary, thesaurus, atlas, concordance, index, pronunciation, biographical information, word and phrase search, and cross-correlation facilities. As textual information is examined, one is able to call up short explanatory notes to assist in understanding the text (e.g. a word's definition), or one can take more lengthy excursions into other parts of the text or into other text files. One may also use electronic bookmarks and marginal notes. These tools can be used in the perusal of any machine-readable text. No human structuring of the text is needed prior to its examination.

1. Introduction

This paper is a report on the Readers Workbench, a research project currently underway at Brigham Young University.

1.1 Workbenches

A workbench is a place where one can perform some specific function. A workbench has tools, workspace, reference material, and storage. Carpenters, electricians, and others use workbenches to perform the tasks involved with their trades and professions. In 1973 the author selected the name "Programmers Workbench"[2] to describe a set of tools he was developing at Bell Labs to assist in program development. Other workbenches developed over the years at Bell Labs include the Writers Workbench, the Instructional Workbench, and the Documenters Workbench[3].

1.2 The Readers Workbench

The Readers Workbench is a set of tools designed to facilitate the reading of electronic text. The Readers Workbench is closely related to the Writers Workbench. The Writers Workbench assists people in generating text by providing information related to writing: style, phrase usage, punctuation, sentence structure, etc. The RWB (Readers Workbench) approaches electronically stored text from the other direction. It provides a set of tools to assist the person who is trying to understand and assimilate text that has already been composed and stored in a computer.

The RWB is sometimes confused with CAI (Computer Assisted Instruction) but is quite distinct from CAI. CAI does include machine readable text but it also involves substantial courseware development through an authoring system. Learning units are composed and structured so that the CAI user can progress through a series of steps to demonstrate mastery of a given subject.

In the RWB there is no effort at courseware development. The basic information operated on by the RWB is raw electronic text, such as that generated by a text editor, word processor, or photocomposer front end. The RWB assists the reader of electronic text by providing the definition, pronunciation, or etymology of words; gazetteer for places; and biographical information on people. The RWB has the ability to locate text segments that correlate with certain other text segments, or that contain particular words or phrases, etc. In fact, the RWB is a wide spectrum of tools which scan, analyze, organize, correlate, and structure text.

The RWB is sometimes assumed to be a system for improving or testing reading skills, such as speed reading. Although such facilities could be included in a RWB, that is not the focus of the current development.

1.3 Text versus Data

The term, "text," as used in this paper, describes the type of information found in most books and other documents. Text is composed of words, sentences, paragraphs, sections, etc. It is distinguished from data in that data generally is structured into fields, records, files, data bases, etc.

Text is entered into a computer via a word processor or text editor. Letters, memos, books, and other hard copy are generated from this text with various printing and typesetting capabilities. Data is entered into a computer via a forms processor or data base

management system. The data is used to generate tables, charts, lists, etc.

Data is fielded in the sense that the basic entities are named fields with certain fixed characteristics. For example, telephone book data might consist of a name field, an address field, and a phone number field. Text is unfielded in the sense that one word follows the next, and there is no particular name or function applied to each piece of the text.

Data is structured in the sense that a sets of fields make up records, records make up a files, and so on. Also certain fields in a record can point to or connect to other fields and/or records. Text also has structure in words, sentences, and paragraphs, but the structure is less formal.

Systems that handle data are called data processing or data management systems. Systems that process text we will call text processing or text management systems. The computer industry currently separates these two fields into two disjoint activities. Data management systems are not well suited for text entry while word processors do not handle fielded data properly.

It is the contention of this author that this distinction should eventually blur so that a single system will effectively handle both text and data.

1.4 Sample Interaction

As a way of further describing how the Readers Workbench might work, we will briefly sketch how a typical interaction or session with the RWB might work. The reader logs into the host system and specifies the book he/she wishes to read. Since he/she has previously read part of the book and left a bookmark, the initial page displayed will be where the reader left off during the last session.

As the reader scans the text, he/she encounters a word with which he/she is unfamiliar. A push of the dictionary function key brings up the definition of that word on an overlay window. As the reader continues, the name of a river is encountered, and the reader pushes the gazetteer function key to obtain an overlay describing the location, size, and other data about the river.

As the reader continues, he/she encounters a paragraph on a topic of high interest. He/she pushes the notes key and examines three notes that have been generated by other readers elaborating on the paragraph.

He/she then decides to interrupt his/her sequential pursual of the text temporarily and delve more deeply into the subject covered by the paragraph. First he/she asks for paragraphs in the same book that are highly "correlated" with the paragraph of interest. Several other paragraphs are displayed from the book in order of correlation measure (based on word frequency counts and thesauri matching). None of the paragraphs appear to relate directly to the reader's interest, so a correlation search is requested for other books. (This probably involves a link to a larger data bank.) Several books are listed and the reader selects one and spends a few minutes reading a portion of it. He/she then returns to the original book and adds his own note to the paragraph.

2. Evaluation of Concept

In section 1 the concept of a Readers Workbench was presented.

In this section the rationale and justification for a Readers Workbench will be presented. Topics that will be covered include: the availability of electronic text, the need for text management, emerging RWB systems, technical breakthroughs, problems in using electronic text, advantages in using electronic text, and finally, conclusions concerning electronic text.

2.1 Availability of Electronic Text

There is a vast body of electronic text already available for processing by systems such as the Readers Workbench. This machine readable text has come from such sources as:

- word processors
- personal computers
- computer-based photocomposers
- electronic mail
- optical character readers
- manual text entry
- etc.

The volume of this unfielded text is probably now comparable in size to that of fielded data files on computers.

2.2 The Need for Text Management

For fielded data we have developed a large number of sophisticated data base management systems. It is suggested that electronic text has great potential value to users also. Thus we need text processing tools as much as we need data processing tools and we need text management systems (beyond word processors and editors) as much as we need data base management systems.

2.3 Emerging Systems

In disciplines where the need for a Readers Workbench has been particularly pressing, we have already seen some specialized commercial products developed. Lexis and Westlaw provide some rather elementary selective reading access tools for court cases. Dow Jones and Nexis provide tools for accessing economic and financial text. Source, CompuServe, Plato, BRS, and Dialog provide access to text on various other subjects. The Writers Workbench is a Bell System product which approaches text from the writer's viewpoint, not the reader's. These systems seem to confirm the fact that text management is now coming of age just as data base management did 10 to 15 years ago.

2.4 Technical Breakthroughs

One of the technical barriers that has kept us in the past from storing much raw text on line has been the cost of secondary storage. Even large expensive main frames were able to store at most the equivalent of several hundred books on line. However, the dramatic decrease in winchester disks over the past 2 to 3 years now makes it possible to have 10 to 100 books on line to a microcomputer (assuming 1 M byte = 1 book). Even more exciting is the prospect of an inexpensive 1 G byte compact laser disk which could provide the text for 1,000 books to a microcomputer (even without networking).

The rapid development and acceptance of local area networks (LAN's) provides a further option: a text management node, similar to a data base server, with the text of thousands of books available

to any workstation on the network.

2.5 Problems with Electronic Text

In addition to the secondary storage issue there are a variety of hurdles that we must overcome before electronic text has any hope of rivaling the printed page. Let us point out a few reasons why hard copy books are still the preferred medium over screens:

- poor resolution, eye strain
- small screen size (24 X 80)
- no figures, pictures, charts, etc.
- non-portable, awkward screens
- necessity of a link to a computer
- no marginal notes, bookmarks, underlining
- unfamiliar, unreliable equipment
- inability to thumb, jump, compare, etc.

There are, however, efforts underway to rectify each of these problems, and it is fair to assume that in the not too distant future, most if not all of these deficiencies will have been resolved.

2.6 Advantages of Electronic Text

On the other hand there are a number of significant factors which are pushing us toward electronic text as the primary medium of communication. Certainly the ease of update and the currency of information will be a factor. Also the ease of transmission and distribution must be considered. As paper and printing costs continue to climb, electronic distribution will become more and more attractive. Also the possibility of almost instantaneous transmission will be an important advantage in many situations.

However, the opportunity for logical processing of the text to assist the reader in understanding and assimilating the text will undoubtedly be the most significant factor in the long run. Having a computer to assist in scanning, analyzing, organizing, correlating and structuring the text will provide a myriad of new tools and functions to make reading easier.

2.7 Conclusions Concerning Electronic Text

In this section we have pointed out that there is a tremendous amount of text that is already in a machine readable form, but that text processing tools are still in their infancy. New disk technologies are going to make it possible to have the text for thousands of books available to each personal computer. If we can iron out some of the interface difficulties that make reading text on a screen unattractive, the positive aspects of text reading via computers will provide significant benefits.

3. Readers Workbench Tools

In this section we will discuss some possible tools that might be made a part of a Readers Workbench. One of the major problems in trying to conceptualize such tools is what might be termed Gutenberg thinking. As in other computer applications, there is a strong tendency to design computer systems so that they mimic the manual (in this case the hard copy) approach. This facilitates the conversion process but often does not take advantage of the full

capabilities of the computer and the exciting new approaches possible only on a mechanized system.

We will first discuss the various techniques that a reader uses when reading traditional hard-copy text. We will then try to break out of the Gutenberg rut and suggest some representative new techniques that could be used in a Readers Workbench. We have divided this discussion up into six parts: determining the essential message of a text, enhancing comprehension, selective reading of text, moving from one text to another, enhancing communication, and individualizing the reading process.

We will summarize which of these techniques are currently implemented in the Readers Workbench in Section 5.

3.1 Determination of Essence

Readers often approach text with the goal of determining the basic message or point of that text without completely reading it. To accomplish this the reader might examine the outline, table of contents, or introduction. Perhaps, one might look at the first or last chapter. Alternatively, one might thumb through the book noting a word or sentence here and there.

All of the above techniques should, of course, also be available in an on-line text reading system (Readers Workbench). In addition a reader's workbench could do the following things. The reader might be provided various statistical reading levels like the ones generated by the Writers Workbench [3]. This would allow the reader to determine if the text is a suitable level for him/her to read.

In addition, frequency counts and distribution profiles might be provided on words and phrases. These would augment the table of contents and index in helping to determine if a topic is covered in the text and, if so, where.

3.2 Enhancement of Understanding

Readers generally have various reference books handy to assist them as they read. Perhaps a word is encountered for which the meaning is not known. Maybe one would like to know where a town is located, or who a person is. The reader does not intend to deviate from the sequential flow of the text he/she is reading, but only to obtain some brief explanation to assist in understanding that text. Typical reference books found in homes and offices, and used for this purpose include:

- dictionaries
- thesauri
- encyclopedias
- gazetteers
- atlases
- biographic references

The reader could, of course, continue to use hard copy reference material in the reading of electronic text. However, a properly designed reader's workbench could also provide such information through a window overlay or some other suitable display mechanism.

Some of the advantages of on-line reference material include:

- currency of information

ease of look-up
 no searching for a book or page number--just point to
 text and hit function key
 further logical processing (e.g. personal vocabulary list)

One problem with this type of RWB facility is the amount of storage needed for a suitable reading reference library (probably 10-100M characters). This exceeds the local storage capacity of most of today's workstations and microcomputers. However, it appears that adequate local storage capacity will soon be available (Section 2). Also networking developments are making it feasible to quickly access centralized reference facilities.

3.3 Selective Reading

One approach to reading is to start at the beginning of a book and to read it sequentially page by page. Alternatively one might decide at the outset to locate only those portions of the book(s) that are of interest and read them. The trick is in locating the pertinent sections. Traditionally to do this we resort to card catalogs and bibliographies to locate the books of interest and then use tables of contents or indexes within the book to find the sections of interest. Sometimes we just randomly scan the book shelves or thumb through the pages hoping to stumble onto something of significance.

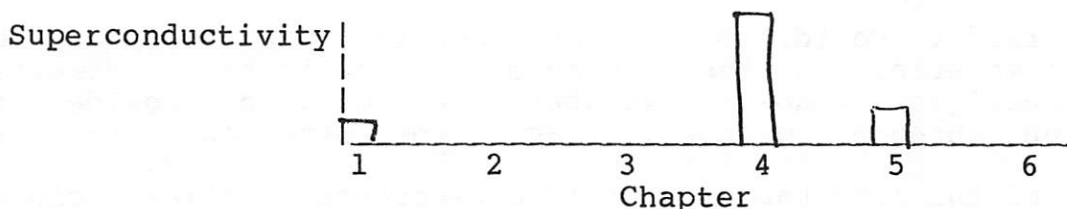
We have experimented with four tools in the reader's workbench which assist in locating the text of interest: boolean queries, citation/author links, word/phrase profiles, and measures of relatedness.

The boolean query is familiar to most computer people and is used widely in data base access. For text access one typically augments the and/or/ not functions with a "within" function. For example, on-line legal systems might allow the query, "assault or battery within 8 words of minor."

One major problem with the boolean query is that of synonyms. We might be thinking of one word but the text contains one of its synonyms. We have started to develop an electronic thesaurus with measures of synonymity and context clues, but have not progressed very far on it yet.

Citation and author links allow one to move to another document cited by, citing, or co-citing the document(s) at hand. An author link is established to documents written by the author(s) who wrote the document(s) at hand. Both of these tools are applicable to documents, but not text segments in general, and require sophisticated matching schemes which have not been fully developed yet.

A word profile chart might also be of value in locating the text of interest. For example, the distribution of the term, "superconductivity", in a book might be:



Based on an interest in this subject, the reader might want to

try to scan chapter 1, but then skip to chapter 4 to do a more detailed analysis of the text.

The measure of relatedness technique allows one to move from a current text segment of known interest (book, chapters, paragraph, etc.) to a set of other text segments that are related through various information-theoretic measures based on word frequencies, etc. This has been shown to be quite effective for document retrieval, but is still being tested for smaller text segments.

3.4 Excursions

As we sequentially traverse some text, we might come across a reference that we would like read or a topic that we might want to delve into more deeply. We thus temporarily leave the book we are currently reading and go to another book or to another section of the same book. Generally we plan to return to the original book and pick up at the place where we left off at some point in the future. The new book we go to may trigger a further idea, sending us to yet another book. This recursive descent may go to six or seven levels.

To allow us to return to the book we started with we might insert a bookmark or perhaps leave it open and stack the new book on top of it. Some type of electronic bookmark is needed in the Readers Workbench to allow this interruption or excursion capability. The tools described in Section 3.3 could, of course, be used to locate additional text on a given topic if the original text does not provide adequate references.

3.5 Enhanced Communication

In Section 2 we alluded to some of the communication weaknesses in current hard copy publication. The flow of information is almost completely one-way. (How many readers call or write the author of a book?) There is also generally little or no interaction between the readers of a book. And by the time a book is typeset and distributed the author had often changed his views.

In the Readers Workbench we have developed a note making capability much like the marginal notes that one makes in books. The major difference with this tool and a hard copy marginal note is that these electronic notes can be sent back to the author, and also shared with other readers if the note's author so specifies. Thus, an electronic book can be an evolving and growing body of information with communication flow between all the participants, and not just a one shot dissemination of one author's ideas. In this regard it is very similar to the Notesfile system developed at the University of Illinois and we are still evaluating how that system might be incorporated into the Readers Workbench[1].

We are also investigating various techniques for allowing a reader to superimpose on a book his own structure and view of the information contained therein. This facility is related to the note facility, but adds to that facility a structure.

3.6 Individualization

The reader of traditional hard copy text individualizes that text by such things as marginal notes, bookmarks, and underlining/highlighting.

The Readers Workbench can provide the electronic equivalent of all these. In addition such things as an interest profile, vocabulary profile, skill level, etc. can be kept to further identify the reader. An interest profile would be of value for the

selective reading functions. A vocabulary profile would help with the reference tools. It is conceivable that the format, speed, and actual content of displayed text might eventually be individualized to each particular reader.

4. Implementation Approach and Status

In this section we will briefly discuss the design approach being taken, the text files being used for testing the system, and the status of current development efforts.

4.1 Design Approach

In the spirit of UNIX our goal is to make the Readers Workbench easily extensible so that new tools can be added by us or others with relatively little difficulty. In some areas such as with the reference material it should be possible to add a keyed reference file in the proper format and define a new function key and be up and running. In other areas we are struggling with the extensibility issue.

A second goal is to allow a wide variety of input/output devices to serve as interfaces to the user. For example, we hope to be able to allow the pronunciation function to come out on a voice synthesizer (e.g. Dec Talk) or on the screen in a format similar to that format in a dictionary. To identify locations we hope to be able to provide maps on a graphics screen or gazetteer descriptions on a character terminal.

A third goal relates to efficiency. This is especially important for the boolean search and correlation tools. Our current approach is to do a complete inversion of the text so that we have an index of the location of each word (less a few low information content words). This almost doubles the storage space required, but greatly speeds up access time for the selective reading tools.

A final goal is to keep the system in the public domain and solicit help in the development of appropriate tools.

4.2 Test Data Bases

We have selected three different sources for text to do the initial testing on:

1. Books published by the Brigham Young University Press
2. The Scriptures
3. UNIX documentation

Sources 1 and 2 have taken a fair amount of translation and reformatting work but are now in use. The BYU Press is using an old DEC-SET front end composition system which had some strange tape formats. The Scriptures were on an IBM system and in a record format with no lines. The UNIX documentation was the most accessible but we have done the least testing on it.

4.3 Current Status of Tools

We currently have the following tools in some working state:

1. Concordance (We have done some interesting comparisons with Strongs Exhaustive Concordance of the Bible, manually generated by many scholars over a period of several years)

2. Boolean Query--nothing unusual here
3. Marginal Notes--This tool works but would benefit from the "Plan to throw the first one away" problem. As noted we evaluated the Notesfile System.
4. Bookmarks--Straightforward
5. Reading Level--Temporarily borrowed from the Writers Workbench
6. Dictionary, gazetteer, etc.--We have a preliminary overlay facility for this, but are looking for a good windowing system.
7. Measures of Relatedness--An early version of this has been tested.

5. Conclusions

Although our early experience with the Readers Workbench is positive, it is still too early to properly evaluate the utility of each of the tools.

In particular we need to expand our text and reference file base and the set of tools available, and to find people who are willing to help develop and test it.

Our plan is to keep it as a public domain system distributed through Brigham Young University. However, we invite others to contribute additional or improved tools, text files, and comments.

REFERENCES

- [1] Essick, Raymond B. IV and Rob Kolstad. "Notesfile Reference Manual." Department of Computer Science, University of Illinois at Urbana-Champaign, Feb. 14, 1983.
- [2] Ivie, Evan L. "The Programmers Workbench--A Machine for Software Development." Communications of the ACM, October 1977.
- [3] "The Writers Workbench," "The Documenters Workbench," and "The Instructional Workbench." (A full set of documentation is available from Bell Laboratories.)

Circuit Design Aids - CDA
A Printed Circuit Board Manufacturing System

T.C. Slattery
Ensign William McCool

Computer Aided Design and Interactive Graphics Group
United States Naval Academy
Annapolis, Maryland 21402

ABSTRACT

CDA is a simple printed circuit board editing and manufacturing system running at the United States Naval Academy to support student electronics projects. The system is designed to be easy to use, run on existing equipment, and simplify the production of student designed printed circuit boards. An Evans & Sutherland PS300 display is used as the interactive device with a VAX 11/780 running Unix as the host processor. Double scale artwork is generated on a flatbed plotter and reduced photographically. Simple artwork can be plotted at 1X scale. Color or black and white verification plots of the artwork can be produced. Numerically controlled milling data is generated to automatically drill the boards before etching.

INTRODUCTION

Circuit Design Aids (CDA) is a simple printed circuit board (PCB) editing and manufacturing system designed for use by students at the United States Naval Academy. The CDA system was developed as a teaching aid for midshipmen in the Electrical Engineering Department who had been designing single and double sided printed circuit boards using manual methods sometimes as crude as pencil on tracing paper. Almost all of these methods required the use of the photographic laboratory to reduce the artwork to 1X scale or to enhance the quality of the original. Due to the small number of boards designed and the setup overhead, turn around time in the photographic laboratory has been approximately two weeks. Each design tended to have its own photographic requirements. Some were modifications of 1X scale designs found in publications. Others were 2X scale done with tape and mylar. Additional time was expended in hand drilling.

A system was needed to produce standard artwork and create drilling data files for computer controlled milling machines. Ideally, the artwork should be 1X scale to bypass delays in the photographic laboratory.

BACKGROUND

Several commercial and public domain printed circuit board design systems were evaluated and a set of requirements developed. The commercial systems were fairly complex and were targeted toward engineers

who could devote the requisite training time. These systems also ran on various computers with proprietary operating systems. It was not clear that any of these systems could be easily integrated into the existing graphics and manufacturing facilities.

Of the public domain systems, "The Electronics Engineer's Design Station" reported by Bering [1] looked the best. Unfortunately, that system had been replaced with a more modern version for gate array and VLSI design. VanCleemput [2] reported on a system at Stanford which had subsequently become a commercial product. The other systems studied [3,4,5] were proprietary. Each system addressed only the design stage of making printed circuit boards; no mention was made of the manufacturing process.

As a result of this research and input from interested professors, a set of requirements was developed for a student PCB design system.

- 1) The system must be easy to use and require a minimal amount of training. Time demands on midshipmen are severe so the system should allow them to do useful work almost immediately.
- 2) All but the most complex artwork should be usable at 1X scale. Eliminating the two week delay of photographically creating 1X artwork would enhance course content and increase productivity.
- 3) The system should integrate with the existing manufacturing facilities. The Naval Academy Technical Support Department has three incompatible CNC milling machines. The hole drilling information should be capable of driving any of the three machines.
- 4) The system should be low cost. High cost capital equipment requires a two to four year procurement cycle. The lower the cost, the quicker the procurement.
- 5) The system should easily integrate with existing computer facilities. The concerns included items such as file transfer, use of existing plot devices and support staff training.

Based on the above requirements and a market survey, the decision was made to create a simple system in house. This would give the students access to a system much sooner than otherwise possible. Their usage would undoubtedly modify the initial requirements. The modified requirements could then be applied to enhance the in house design or to select an appropriate commercial package.

The Naval Academy had already purchased two Evans & Sutherland PS300 calligraphic workstations and an Ikonas RDS-3000 color frame buffer. The capabilities of the PS300 workstations are sufficient to support the type of design system proposed.

SYSTEM DESCRIPTION

CDA was first prototyped during the summer of 1983. Figure 1 shows the hardware system configuration. The central software component of the system is the printed circuit board editor 'WIRE'. Post processing programs support checkplots on various devices, creating CNC (Computer Numerically Controlled) mill hole drilling data and recovery of data after crashes. The available functions are shown in Figure 2.

System Configuration

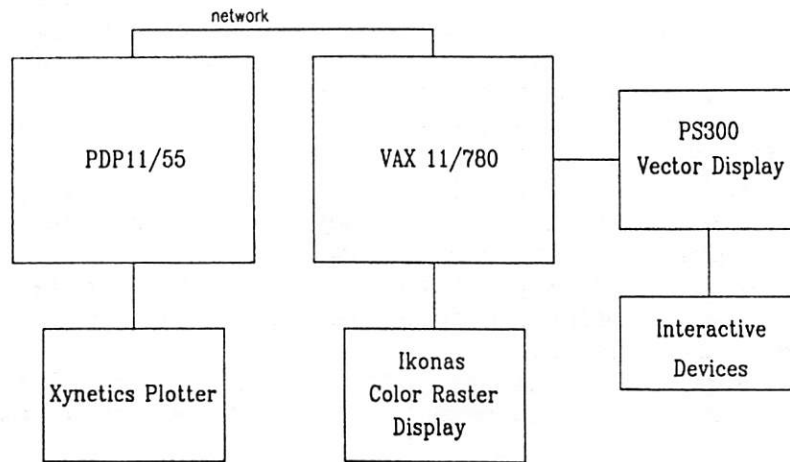


Figure 1

CDA System Functions

WIRE

- * Toggle layers on & off
- * Zoom & pan
- * Add lines, dips, vias, and connectors
- * Delete objects
- * Save and retrieve designs

Post Processors

- * Check plot generation
- * Artwork generation
- * CNC mill information generation
- * Crash recovery

Figure 2

The prototype version is being evaluated by the Naval Academy staff. A production version is being designed based on the results of these evaluations.

CDA separates interactive graphics processing from background processing. The use of 'smart' graphics devices such as the PS300 and RDS-3000 relieves the host of much of the real-time or calculation intensive demands that frequently accompany interactive graphics. A PS300 resident program supports the interactive devices (tablet, function keys and control dials) used to position objects on the screen, to control zooming and panning and to point at objects. Communication with the Unix host is only necessary when a change to the host database is required; e.g., when objects are deleted or added. This update information is generally compact enough that a 9600 baud serial communications line is sufficient. Since the host programs do not have to support the interactive functions, they are much smaller and more efficient.

OPERATION

The interactive layout is performed on a PS300 workstation (Figure 3). Various functions are selected by the twelve LED labeled function keys.



Figure 3 - PS300 Workstation

A ten inch square design grid (Figure 4) is used to aid in placing components and lines.

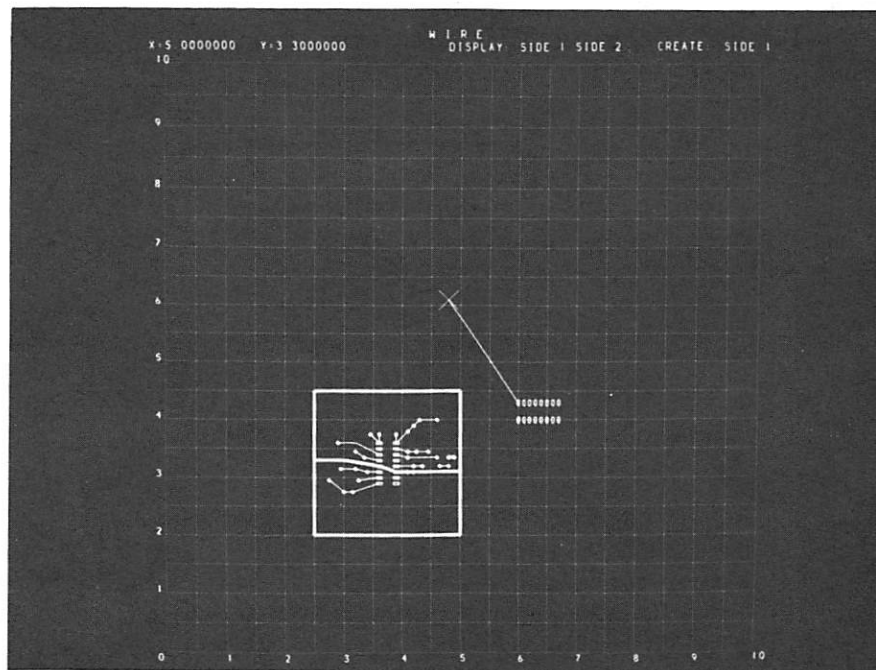


Figure 4 - Design Grid

Using the function keys, the user may select ADD LINE, ADD DIP, DELETE, SAVE, etc. Some functions drop to a sub-menu where the LEDs change to reflect the new set of choices. For example, selecting DIPs causes the function key labels to display the range of DIP package sizes available. Pressing the appropriate function key selects the desired package size. Then, using a tablet, the user moves the component to the desired grid location for placement. Depressing the button on the tablet mouse causes the host to be informed of the placement and the object is added to the data base and the display list. All items are aligned on a .05 inch grid. Deletion is selected in a like manner and the tablet is used to "pick" the object to be removed. Again, the host is informed of the modification and the data base and display list are updated. Lines are added to one of two layers and all other items appear on both layers. The system provides LED labeled dials for zoom and pan so that dense boards may be easily designed. Additional functions allow the user to save or retrieve designs. Since the existing workstations utilize monochrome displays, a pair of functions independently toggle each of the layers on and off to reduce viewing congestion.

ARTWORK VERIFICATION AND GENERATION

CDA is capable of creating three types of artwork:

- Monochrome checkplots.
- Color checkplots.
- Final artwork generation.

Two types of monochrome checkplots are produced on a Tektronix 4014. One type shows both sides of the board at once. This is adequate for simple or single sided boards, but is confusing for complex double sided boards. The other output shows one side at a time (Figure 5). These plots have the obvious disadvantage of not being able to easily determine layout conflicts.

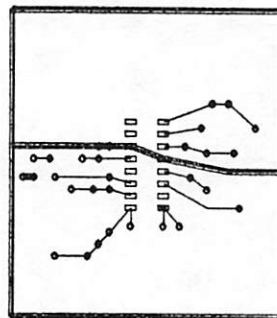


Figure 5 - Tektronix Check Plot

Color plots created on the Ikonas frame buffer provide a fast multilayer checkplot. This output is used for quick checks during design. The addition of a color camera or copier will allow fast color hardcopy. Checkplots are currently created on a Xynetics flatbed plotter. Three colors are used: red for side 1, blue for side

2 and black for the through-holes.

The final artwork generation is done on the Xynetics plotter at 1X, 2X or 4X scale. Each side is plotted separately with black ink on mylar. The 2X and 4X scale artwork is reduced photographically. Experimentation with producing 1X scale artwork for direct use in the manufacturing process has been encouraging. This procedure eliminates the two week photographic laboratory time for these boards.

MANUFACTURING

An advantage of CDA is automatically generating the milling data necessary to drive the CNC milling machine used to drill the board. CNC milling is important for plated-through holes since the board must be drilled before it is etched and plated. The holes must also be clean and without burrs for the plating to succeed. Drilling by hand is impractical because of the tight tolerances in feed rates necessary to keep from burning the board, dulling the drill bit or creating burrs on the copper plating. It is also very difficult to accurately position the board manually. Post processing routines create checkplots of the drilling locations (Figure 6) and generate the milling data. The data is then sorted for more efficient drilling.

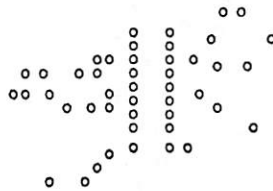


Figure 6 - Hole Location Checkplot

The final CNC data is downloaded from the host by a machinist using a Tektronix 4051 desktop computer. The CNC data is stored on the 4051's internal magnetic tape so that successive drilling operations do not require any further intervention on the part of the host computer. This data is converted by the 4051 to machine specific milling commands which are passed to the CNC mill controller via the Tektronix General Purpose Interface Bus. The drilling time for 100 holes is ten to fifteen minutes; two boards can be drilled on each pass. The drilled board is then plated and etched using Naval Academy or commercial facilities.

FUTURE DEVELOPMENT

The system requires further development based on suggestions made during testing and evaluation. Fraser's [6] Unix Circuit Design System offers the potential to produce the schematic of the project, place components on the board and create connect net lists which can potentially drive an auto-router, connection verifier or both. Rat nest plots and line density histograms would help in placing devices at this stage.

The PS300's vector capability makes placing objects, zooming, panning and rubber-banding much more interactive than on most raster

devices. However, boards with large ground plane areas are not easily accommodated. Methods of cross-hatching these fill areas are being considered to solve this problem.

For many of the simpler boards it may be possible to display the artwork on the Ikonas frame buffer at 1024 X 1024 pixel resolution and produce full scale artwork with an attached camera using Polaroid transparency film. Experimentation in this area is planned.

A fourth 'layer' is needed to represent the silk screen labeling. Shliferstein [5] reported on the use of a color system to provide 2-1/2 dimensional information for PCB work. The addition of a color PS300 display would provide the additional 1/2 dimension.

Block copy, move and delete commands are needed to handle digital designs where repetitive areas occur.

CONCLUSIONS

The graphics editor is an improvement over manual (tape and mylar) methods of board design although it may be less convenient for the students since they must come to the graphics laboratory to create the artwork. The improvement comes from faster layout editing and the fact that lines are easier to delete graphically than manually. It is easy to save intermediate designs at critical decision points so that if a potential layout doesn't work, the student may easily backtrack. Preliminary sketches and analysis coupled with interactive editing aids in obtaining optimum component placement.

The manufacturing process has been greatly improved. Standard size artwork reduces the setup time in the photographic process. Full size (1X scale) artwork capability has completely eliminated this delay in many cases. Automatically drilling the boards has eliminated hand drilling and added the capability of producing plated-through holes.

REFERENCES

1. D.E. Bering, "The Electronics Engineer's Design Station". Proceedings of 17th Design Automation Conference, 1980.
2. W.M. VanCleemput, K.R. Stevens, T.C. Bennett, J.A. Hupp, "Implementation of an Interactive Printed Circuit Design System". Proceedings of 15th Design Automation Conference, 1978.
3. F.D. Skinner, "Interactive Wiring System". Proceedings of 17th Design Automation Conference, 1980.
4. H.C. Bayegan, E. Aas, "An Integrated System for Interactive Editing of Schematics, Logic Simulation and PCB Layout Design". Proceedings of 15th Design Automation Conference, 1978.
5. Abe Shliferstein, "Experiments Using Interactive Color Raster Graphics for CAD". Proceedings of 19th Design Automation Conference, 1982.
6. A.G. Fraser, "Circuit Design Aids", The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978.

A TRANSITION DIAGRAM EDITOR

Charles C. Mills

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720 USA

Anthony I. Wasserman

Medical Information Science
University of California, San Francisco
San Francisco, CA 94143 USA

1. Introduction and Background

This paper describes the design and implementation of the Transition Diagram Editor (tde), a graphics editor for making directed graphs, with boxes, circles, arrows, and text, for the RAPID/USE[1] system for rapid prototyping of application user interfaces. Tde and RAPID/USE are part of the User Software Engineering (USE) methodology[2], along with the Troll/USE relational database management system[3], the PLAIN programming language[4], and the Interactive Development Environment (IDE). The USE tools run under 4.2 BSD Unix. Tde currently runs on the SUN workstation.

User Software Engineering is a methodology and supporting environment for the development of interactive information systems. The USE methodology recommends the creation of prototype versions of the user/program dialogue as a way to experiment with the user interface and to quickly build a working version of a system.

The USE model of an interactive information system (IIS) includes a set of augmented state transition diagrams, which describe the flow of control through an interactive dialogue. The entire interactive session is termed a *conversation*, which may be divided into several *subconversations*. The set of all possible conversations is modeled by one or more USE transition diagrams.

In its simplest form, a USE transition diagram contains four different symbols, as follows:

- (1) *node* -- shown by a circle, representing a stable state awaiting some user input. Each node within a diagram has a unique name, and an output message may be displayed when a node is reached. The message associated with a node may contain display formatting commands that control cursor address, standard/inverse video, clear screen/line, etc. Those commands are specified in a simple, readable, device-independent language. One node is designated the *starting node*, represented by two concentric circles, and there is a single exit node.
- (2) *arc* -- shown by an arrow, connecting nodes to one another. Each arc represents a state transition based on some input. The input is designated either by a string literal, such as 'quit', or by the name of another diagram, enclosed within angle brackets, such as <diag2>. One arc emanating from each node may be left blank, in which case it becomes the *default* transition, and is taken only when the input fails to match that specified on any of the other arcs.

- (3) *action* -- shown by a small square with an associated integer or function call. An action may be associated with a transition to represent an operation that is to be performed whenever a specific arc is traversed. The same action may be associated with more than one arc.
- (4) *subconversation* -- shown by a rectangle, in place of a node, labeled with the name of a subdiagram enclosed in angle brackets. A subconversation causes an invocation of the named subdiagram. The calling conversation resumes on exit from the subdiagram. Thus, the size of individual diagrams can be kept small, readable, and manageable, and a large system may be built using a library of small modules.

The User Software Engineering approach made many extensions to the traditional state transition diagram, including declaration and use of variables, branching after actions, format control directives for output display, and distinction between buffered and unbuffered input. The resulting notation, USE transition diagrams, are still based on finite state concepts, but are extended to meet the needs of specifying and building interactive information systems.

Figure 1a depicts a USE transition diagram, drawn and printed using tde. Figure 1b shows its encoding in the USE diagram specification language. The diagram represents a library management system. It starts by clearing the screen and asking whether the user needs instructions. At the entry node, named **greet**, the message may be read:

clear screen, row 8, center: "Welcome to Library System",
down 4 rows, center: "Do you know how to use the system (y/n)?",

At the entry node, if the user types "n", control passes to node **help**. If the user types "y", action number 12 is called, and then the menu at node **mainmenu** is displayed. If the input does not match the label on any arc leaving the start node, then the default transition to node **err1** is taken, and so on.

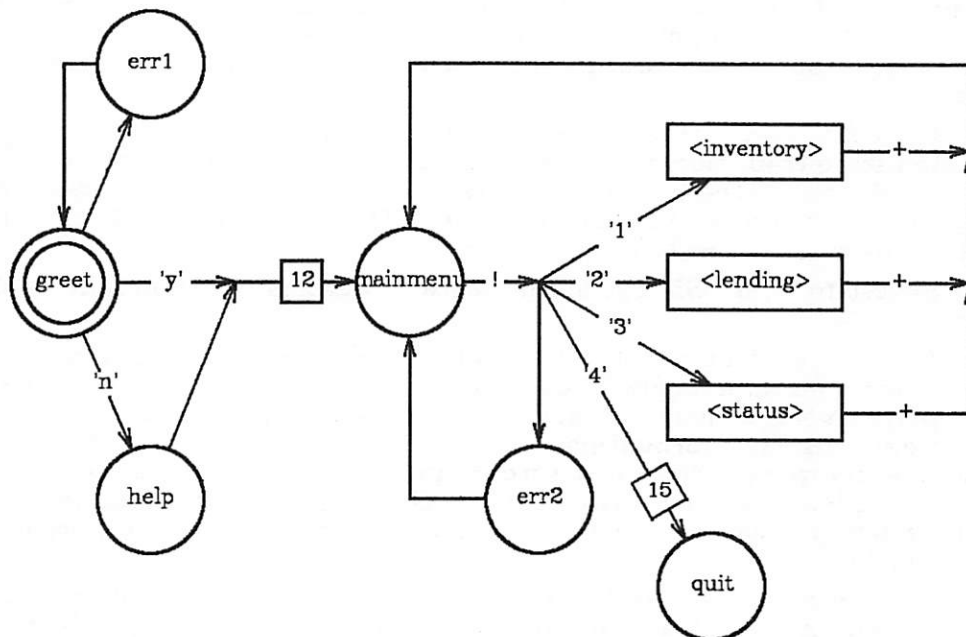


Figure 1a -- A USE Transition Diagram -- the top level of a library management system

```

diagram libsys entry greet exit quit
node greet
    cs,r8,c_ 'Welcome to Library System',
    r+4,c_ 'Do you know how to use the system (y/n)?'
node help
    cs,r3,'Library System is an interactive, menu-driven, system for management',
    nl,'of a small library. It handles the activities of book acquisition',
    nl,'and removal, checking books out to cardholders, checking books in',
    nl,'and inquiring as to the status of books in the library.',nl,
    nl,'The system will prompt for all user input, and use of the system',
    nl,'should be straightforward and self-explanatory. Good luck.',
    nl,'Type RETURN to continue.'
node mainmenu
    cs,hm,r10,c10,'Please choose subsystem',r12,c15,
    '1) Book acquisition/removal',r+2,c15,
    '2) Book lending/return',r+2,c15,
    '3) Status inquiry',r+2,c15,'4) Quit',
    r+2,c10,'Your choice (1-4): '
node quit
    cs,r12,'Byebye....'
node err1
    r$,'Please type "y" or "n". Press any key to continue.'
node err2
    r$,'Please type a number between 1 and 4. ',
    'Press any key to continue.'

arc greet
    on 'n' to help
    on 'y' do 12 to mainmenu
    else to err1
arc help
    else do 12 to mainmenu
arc mainmenu single_key
    on '1' to <inventory>
    on '2' to <lending>
    on '3' to <status>
    on '4' do 15 to quit
    else to err2
arc <inventory>
    skip to mainmenu
arc <lending>
    skip to mainmenu
arc <status>
    skip to mainmenu
arc err1
    else to greet
arc err2
    else to mainmenu

```

Figure 1b -- Textual Encoding of the Library System of Figure 1a

RAPID/USE is a diagram interpreter. It parses the textual encoding of a set of diagrams, builds run-time tables, and runs the prototype system. Using diagrams to show the flow of control of a program is very helpful for designing and debugging. However, the labor-intensive bottleneck in designing a system with this technique is the process of drawing diagrams and encoding them in their textual form. The Transition Diagram Editor was conceived as a tool to automate drawing and encoding, and thereby to speed up the debug cycle for rapid prototyping.

2. Design

2.1. Terminology

Graph theory speaks of *graphs*, which are sets of *vertices* and *edges*. In USE Transition Diagram terminology, as is used in this paper, these are *diagrams*, *nodes*, and *arcs*. An *arc* is an edge in a graph unless the context suggests otherwise. (A *circular arc*, has nothing to do with graphs, but refers to a piece of a circle.)

A *vector* usually means a straight line segment drawn on a graphics device. A *raster* is a rectangular array of data defining a rectangular image on a raster graphics device.

An arc (edge) is displayed as a series of connected straight lines, called *arc segments*. The term *vertex* is usually used here to refer to the point where two or more arc segments meet.

2.2. Capabilities

The editor draws circles, boxes, arrows and text on the graphics display in response to user commands, and generates the USE transition diagram specification language. It knows about the connectivity of the graph it is manipulating, so it automatically takes care of most of the details of formatting the drawing, allowing the user to concentrate on the semantics of the diagram. Arcs are defined as relationships between nodes. To create a new node the user specifies (by pointing with the mouse) the place in the diagram to put the node. To create an arc, the user simply points to the nodes to be connected. The editor knows where to draw the lines and arrowheads. If the node is later moved, the arc is redrawn automatically.

An editor that is part of a programming/debugging cycle must introduce as little delay as possible into that cycle, so tde is designed to be fast. The user is able to build graphs almost as quickly as he can move the mouse, click the buttons, and type.

Desirable capabilities in a 2-D graphical editor include translation, rotation, and scaling of arbitrary subsets of the elements of the picture, and scrolling, panning and zooming a viewing window onto the picture. Of these, tde provides only translation, although the implementation under the SUN Window System will have windowing without zooming. This limitation does not create problems for drawing USE transition diagrams, because USE provides an abstraction facility: any node or arc may represent a call to a subdiagram. The methodology encourages the designer to use hierarchies of small readable diagrams. However, the inability to scale and zoom is likely to limit the generality of the editor: graph browsing and editing applications that must deal with large complex graphs will find tde in its present form to be inadequate. The emphasis in this project has been to show the usefulness of quick response and maximal feedback in an interactive, graphics-driven, programming system.

2.3. Graphical Objects and Data Abstractions

The USE Transition Diagram consists of circular nodes, concentric circular start nodes, rectangular subconversation nodes, names on the nodes, messages associated with the nodes, arcs between the nodes, labels on the arcs, and square action boxes on the arcs. Although provisions must be made for browsing and

editing the text of a node message, the message text will not normally be displayed in the diagram; there usually will not be room. When the user wants to see or edit the message, a window may be opened for that purpose.

Arcs are built with a series of one or more straight line segments, with the last segment terminated by an arrow head. Although, strictly speaking, an arc is a relationship between nodes, and arcs are independent of one another, it is often convenient to draw branching arcs and joining arcs to avoid drawing many long parallel arcs. Each arc has an arrow head at its destination end, and where three or more line segments meet to form a branching or joining arc, arrow heads are drawn on the incoming segments to make the branching or joining nature of the intersection clear. Branching and joining is limited in that it is illegal to create a cycle within an arc by branching from an arc and rejoining the same arc at an earlier point. Thus, it is always possible to interpret a diagram strictly as a graph, where arcs are simply relations between nodes; branching and joining are only a drawing shorthand. The editor decides the details of how to attach a node to an arc: where on the node to start or end the arc, and where to draw the arrow head. The user need only specify the positions of the nodes and the arc vertices. Dangling arcs are illegal: an arc must start at a node and end at a node; again, an arc is always interpreted as a relation between nodes.

Nodes and *arcs* might be natural elementary data objects in a graph editor. But because arcs consist of one or more straight line segments, and any arc segment may be shared by more than one arc, *nodes* and *arc segments* were chosen as the elementary data items. The point where two or more arc segments meet, which we will call a *vertex*, is represented internally as a special type of node. Thus, the positions of all nodes are specified by the user, while the position of an arc segment is dependent on the positions of the two nodes that it connects. All the other objects in the diagram are attributes of nodes or arcs. Attached to nodes are labels and messages. Attached to arcs are labels and action boxes.

Each arc segment knows which nodes it is connected to. Each node has pointers to doubly linked lists of the arc segments connected to it. Some of these links are redundant, saved for efficiency. The position and orientation information in the arc segment records is also redundant. It can always be computed from the positions of the source and destination nodes. But it is saved for speed in picking, erasing and redrawing.

2.4. Command Syntax

The pioneers of mouse and menu interfaces at Xerox PARC insist that command syntax should have a postfix style: first, a list of points or objects in the display is selected (the operands), then a command is selected from a menu (the operator). Two advantages of postfix syntax are that it is *modeless* and that it is easy to implement and extend. However, there are some things that cannot be done with a postfix style. An operation that provides continuous real-time feedback does not fit into this scheme. An engineering workstation with its frame buffer on-board makes it possible to redraw parts of the display fast enough to provide continuous feedback. The value of such feedback in an interactive editor can be great. As an example, consider the **move** command, to translate a node from one place in the diagram to another. Using postfix syntax, the user first selects the object to be moved, then selects two points in the picture that define the vector (distance and direction) to be added to the object's current position. Finally, the user selects the **move** command, and the editor erases the object and redraws it at the new position. If that position is not quite right, as is frequently the case, the command sequence must be repeated, perhaps several times.

Contrast these user actions with what is possible if the hardware allows a high bandwidth to the display, and if we forego postfix syntax. First select the **move** command. Next select the object to be moved. Now move the mouse across the display as the picture is continuously redrawn to show the new position. Adjust the final position until it looks just right. The paradigm here is a human hand: it picks

up the object, moves it to its new place, and puts it down. A real human hand is controlling the mouse, so the action is completely natural. The operation can always be done perfectly the first time, with a minimum of user action. Many operations have been compressed into a single one.

With this high level of feedback, different *modes* for different commands are no longer confusing. The considerable value of such feedback in an editor justifies the extra programming effort that it entails.

2.5. The Menu

SUN provides a window system software package[5] that includes entry points at all levels, from bit-blit and vector drawing primitives through virtual terminals in overlapping windows. This package includes primitives for pop-up menus. However, the window system had not been released with the workstations that were available at the time this project was started. Currently, tde permanently displays a fixed menu at the left side of the screen.

Tde marks the current command by highlighting the item in the menu with reverse video. In addition, tde provides feedback by highlighting a menu item whenever the mouse cursor points at it, so the user knows for sure before clicking the button which item will be picked. Figure 2 shows a raster image of the tde menu, with one item highlighted.

2.6. Commands

The SUN mouse has three buttons. The left button is used for selecting a menu item or object or position in the diagram. The center button undoes the previous command. The right button redraws the screen. Here is a command summary.

start, circle, rectangle

These commands insert a new node, which is drawn at a position selected by the user, and optionally dragged to a new position.

arc Insert a new arc (a series of one or more arc segments) between existing nodes, arc segments, or vertices. The user selects the source object, optional intermediate positions for vertices, and then the destination object.

vertex Insert a new vertex in a selected arc segment, and drag it to its final position.

label Label a node or arc segment selected by the user. The editor prompts for keyboard input. The new label is centered on the arc or node as each character is typed.

action Insert an action box in a selected arc segment. The editor prompts for an action number.

edit msg Append the text message associated with a selected node. The user may escape to a text editor to modify or browse the message.

show msg Display a text message (normally hidden) associated with a selected node.

del msg Delete the text message associated with a selected node.

move When a node or vertex is selected, the **move** command drags it to a new position, and redraws all its connecting arc segments. When applied to an arc segment, this command disconnects the segment from an adjacent node or vertex and reconnects it to another selected node, vertex, or arc segment. **Move** can also be applied to all nodes in a user defined rectangular region in the diagram. The user selects two opposite corners defining the region, then drags the rectangle to a new position. The nodes and vertices with center points inside the rectangle are moved.

DIAGRAM:

```

start
circle
rectangle
arc
vertex
label
action
edit msg
show msg
del msg
move
delete
clear
grid
align 32
read
edit
alt
next
subconv
write
raster
pic
rapid
help
exit

```

Figure 2 -- the menu

delete Delete the selected node, vertex, or arc segment, including all its dependent labels, messages or action boxes. Dangling arc segments are not allowed, so deleting a node also deletes all its connecting arcs. **Delete** may also be applied to all nodes in a user defined rectangular region.

The preceding menu items all require some action with the mouse to specify operands (positions or objects) in the diagram. In contrast, most of the following commands take effect as soon as they are picked.

clear Discard the current diagram.

grid Draw or erase a grid on the background as an alignment aid.

align The editor prompts the user to change the alignment number, a measure of the coarseness of an imaginary grid on which new nodes and vertices are aligned. The current alignment number is displayed next to the **align** command in the menu.

read	The editor prompts for a diagram name, reads nodes and arcs from a file and superimposes them on the current diagram.
edit	Discard the current diagram and edit a different one. The name of the diagram to edit is the current diagram name, displayed in the upper right corner. (See <i>DIAGRAM</i> command below.)
alt	Return to the last previously edited diagram.
next	Edit the next diagram specified on the command line at startup.
subconv	Edit a subdiagram. The user selects a rectangular node with a label enclosed in angle brackets, representing a call to a subconversation.
write	Save the current diagram in a file. The file contains an ascii description of the diagram, specifying node positions, arc connectivity, messages, labels, and actions.
raster	Save in a file a raster image of a selected rectangular region of the current diagram.
pic	Write to a file a description of the image of the current diagram for the pic[6] preprocessor for troff[7]. The diagram in Figure 1a was drawn with tde and printed using pic.
rapid	Generate the USE textual encoding of the current diagram and write it to a file, suitable for input to RAPID/USE.
help	Display a screen-sized summary of commands and syntax.
exit	Quit the editor.

DIAGRAM: <name>

If the user selects the diagram name in the upper right corner of the display, the editor prompts for a new diagram name.

External file names are made by adding extensions to the diagram name: **.tde** for a saved diagram, **.rast** for a raster file, **.pic** for the pic description, and no extension for the USE transition diagram description.

3. Implementation

The SUN software release that was available at the time the project was started provided three interfaces to the graphics display. The highest level interface was SunCore, SUN's implementation of the ACM Siggraph Core Graphics Standard[8,9]. The Core standard is a large, powerful, general set of procedures. Because Core must do arbitrary transformations and work within a user-defined coordinate system, it uses floating point arithmetic throughout.

We will call the second interface provided by SUN the RasterOp level. It consists of the low level device dependent bit-blit routines on top of which the core standard and SUN's other graphics software is built. There are only a few of these routines. They provide the ability to copy rectangular rasters from screen to screen, memory to screen, etc., to draw vectors (straight line segments) and to print text (using fixed width raster fonts). They use integer device coordinates. They are fast (critical sections are written in assembly language). Under the original SUN software release they are documented in an appendix to the SunCore manual. For those familiar with the newer version 1.0 SUN software, this is the Pixrect interface.

For the third interface, at the lowest level, it is possible to map the frame buffer to user memory with the mmap() system call and write directly to the frame buffer.

SunCore was chosen for the initial implementation of tde. ACM Core is a standard of sorts, and, for portability, the value of using a standard interface is clear. But, for several reasons, SunCore proved to be a poor choice. Both drawing and picking are annoyingly slow using SunCore. And the Core package makes many simple things difficult or impossible to do right. So, the initial SunCore version was rewritten to use the bit-blit (RasterOp, Pixrect) interface.

All tde's objects except circular nodes can be drawn using vectors and fixed width fonts. Computing a circle at run-time is slow. In particular, it is much too slow for *dragging* a circular node. The solution is to drag a raster image of the node defined at compile-time.

4. Future Directions

The editor will be made to run under *sunttools*, SUN's virtual terminal system. This conversion will be accompanied by the change to SUN's pop-up menus, for two reasons. First, there will not be room in a smaller window to display a fixed menu. Second, under the window system, reading the mouse will be too slow to be able to highlight commands as the mouse moves through the menu area. (SUN solves that problem for its own menus by locking the screen whenever a menu pops up.) Thus, the pop-up menu redesign will entail reserving a spot in the window where the current command can be permanently displayed. Under *sunttools*, display and editing of the message attached to a node can be done cleanly in a separate window.

Recently RAPID/USE and the USE diagram description language have been augmented in several ways. A notation needs to be defined to represent the new elements of the language, and tde's USE code generator needs to be modified to produce the expanded syntax. The changes are as follows:

- (1) Previously, a transition could trigger an action only through an action number; the user must supply an action routine, written in a procedural programming language, to interpret the action numbers. Now, in addition, an action may consist of a call to a named function or a call to a script of troll/USE database queries, or it may increment or decrement a variable. To handle this, an action box may contain +, for increment, -, for decrement, *do*, for a named function, or *ca*, for a call to a troll/USE script. Associated with an action box will be hidden text containing the name of the variable to increment or decrement, or the function call with its arguments or the pathname of the script. The action box text may be edited or viewed like a node message.
- (2) With the old version of RAPID/USE, the selection of a transition from one node to another was dependent on user input at that node, or on the return value from an action or subconversation. The new version provides a case statement, so that a transition may be chosen based on the value of a variable. In the diagram, this switch will be represented by an arc, labeled with the string *case [varname]*, followed by labeled branches.

Tde does little syntax checking when it generates code for RAPID/USE. When RAPID/USE parses its input and reports a syntax error, it reports the line number from its input file where it found the error. This is inconvenient. The user should be able to completely ignore the intermediate form of the USE diagram description language. Instead of a line number, the object that triggered the error should be highlighted in the diagram. The only complete solution would be to more closely integrate tde and RAPID/USE, or to build RAPID/USE's parser into the editor. Short of that, it should not be difficult to build in several simple checks for common errors, which could be reported at the time the *rapid* command is run, when code is generated.

Currently, the most laborious process in building a prototype system using tde and RAPID/USE is writing the node messages. Trial and error is the only way to polish the format of a menu or form presented by the prototype system to the user. The ability to use the text editor to create the node message should be supplemented with a special purpose what-you-see-is-what-you-get message editor. This program would allow the user to drag blocks of text around the screen with the mouse. It would be possible to manipulate text in reverse video, to specify *clear screen* or *clear to end of line*, etc. The message editor would automatically generate the USE language for node messages.

A few improvements might make the editor useful to a wider range of applications. The editor could provide a greater variety of node shapes: ellipses or ovals, regular polygons, arbitrary polygons, even arbitrary rectangular rasters. The user should be able to define node sizes as well as shapes. Implementation would be straightforward if the user could put up with some sloppiness in node picking; it might be hard to find a fast general node selection algorithm that would find a hit if and only if the mouse is strictly inside an arbitrary user-defined outline. Directionless arcs (no arrows) and arcs with arrows at both ends could be supported. If the editor could zoom it could handle larger graphs; zooming could and should for efficiency be done using integer arithmetic, raster fonts, and quantized scale changes. It would not be difficult to provide the option of drawing arcs as spline curves for hardcopy prints: given the current framework of arcs defined as a series of arc segments, the vertices could act as control points for a variety of spline that is tangent to the control polygon at its endpoints and terminates at the endpoints.

Tde has achieved its primary design goal: it is a powerful tool for designing USE transition diagrams. Its speed and high level of feedback make it easy to learn and use. Although it should do more syntax checking of its diagrams, it shows off its usefulness in the run-time debugging phase of prototype design. A graph of the flow of control of a program is extremely helpful in writing and understanding the code. If graph editors had been available twenty years ago, computer science might have been able to live with FORTRAN and the *goto*!

5. References

- [1] Wasserman, Anthony I. and Shewmake, David T., *A RAPID/USE Tutorial*, Medical Information Science, U.C., San Francisco, San Francisco, California (5/84).
- [2] Wasserman, A.I., "The User Software Engineering Methodology: an Overview," pp. 591-628 in *Information System Design Methodologies -- a Comparative Review*, ed. A.A. Verrijn-Stuart, North Holland, Amsterdam (1982).
- [3] Wasserman, Anthony I. and Kersten, Martin L., *A Troll/USE Tutorial*, Medical Information Science, U.C., San Francisco, San Francisco, CA (5/84).
- [4] Wasserman, A.I., Sherertz, D.D., Kersten, M.L., Riet, R.P. van de, and Dippe, M.D., "Revised Report on the Programming Language PLAIN," *ACM SIGPLAN Notices* 16(5) pp. 59-80 (May, 1981).
- [5] , *Programmer's Reference Manual for the Sun Window System*, Sun Microsystems, Inc., Mountain View, California (3/84).
- [6] Kernighan, Brian W., *PIC -- A Graphics Language for Typesetting -- User Manual*, Bell Laboratories, Murray Hill, New Jersey (3/82).
- [7] Kernighan, Brian W., *A TROFF Tutorial*, Bell Laboratories, Murray, Hill, New Jersey (5/83).
- [8] , "Status Report of the Graphic Standards Planning Committee," *Computer Graphics, a quarterly report of SIGGRAPH-ACM* 13(8/79).
- [9] , *Programmer's Reference Manual for SunCore*, Sun Microsystems, Inc., Mountain View, California (5/83).

Optical Storage Management under the Unix[†] Operating System

Perry S. Kivlowitz^{††}

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, New York 11794

ABSTRACT

This paper identifies and describes the difficulties presently facing Unix installations wishing to provide viable and secure backup facilities. Optical storage disks (both write-once and erasable) are introduced as an extremely promising alternative to currently available mass-storage devices. Several problems associated with the merging of write-once optical disks with the Unix operating system are discussed. Two file system structures (for write-once disks) are proposed. A third file system structure is proposed for use with erasable optical disks. Each proposal successfully addresses the problems introduced by optical disks and should be considered as good starting places for sites wishing to incorporate optical storage technology into their systems. **Note:** This paper deals exclusively with data structures and software to be used with optical disks. Very little hardware related information is contained herein.

In many ways the purchasing of large disk systems can be likened to the way a compulsive charge card user makes use of his plastic money. The compulsive charger sees something in a store that strikes his fancy and immediately plunks down the charge card without stopping to consider how the charge debt will actually be paid off. Similarly, many very large disk subsystems have been purchased without consideration given to how the beasts will be backed up. The realization that (priceless) information stored on (merely) expensive disk subsystems can disappear with the pull of a (down right cheap) plug without recourse to a backup medium is a sobering one indeed. Installations with large disk systems and no backup procedures are living on borrowed time. But, how much better are those installations with backup procedures in place?

It is quite common place for a Unix machine to have upwards of 250Mbytes of on-line disk storage. Until recently there were only four choices as to the medium to be used for backup purposes:

[†]Unix is a trademark of Bell Laboratories

^{††}The author's present address is: Heurikon Corporation, 3001 Latham Drive, Madison Wisconsin, 53713
copyright 1984, Perry S. Kivlowitz

- 1 Increasing the number of disk drives and keeping some portion of these reserved for backup purposes. This is prohibitively expensive and does little more than delay the problem of backup. This alternative obviates the possibility of keeping backups for arbitrarily long periods of time since space consumed by archival storage is a non-reusable resource. Also, the additional (fixed media) drives are susceptible to the same hazards of fire, power glitches, spilled coffee, etc. as the primary data drives.
- 2 Purchasing cartridge type drives or drives with removable platters. These offer a distinct advantage over the purchase of additional fixed media disk drives in that the removable disk packs are exactly that, removable. When full they can be replaced with fresh packs placing no (quickly reached) bound on the amount of information that can be preserved. When removed, the packs could be stored in a fire/flood resistant vault further ensuring their safety.

To their detriment, removable disk packs are bulky and require careful handling and storage. The capacity of removable disk packs are in general not as large as fixed media drives often requiring two to ten removable packs to completely backup a single large fixed media drive.
- 3 Magnetic tapes (either reel-to-reel or cartridge) represent the third alternative and the choice most often taken when information archiving is undertaken seriously. To their advantage, tapes are cheap and can stand considerable abuse. Unfortunately they have a limited life span, their integrity degrades after only a few uses. Moreover, they are slow, cumbersome to use, and can hold only from twenty to sixty megabytes per tape. A full backup to tape (or restoration from tape) of a half gigabyte drive might easily consume twelve man hours and as many as twenty five tapes.
- 4 The last alternative and probably the most common is the floppy disk. Floppies containing from one half to one megabyte are often used for making personal backups of important files. For large scale archival purposes however floppies are unreasonable due to their limited capacity. It should be noted that floppies (as well as magnetic tapes to a larger extent) serve as the principal means of data distribution. Both are fine when the information to be distributed can be contained on a small number of floppies or tapes. If the number grows large however, the ease of distribution drops rapidly while the cost as well as the risk of error increases rapidly.

As can be seen from the above discussion each of the four alternatives suffers from some severe flaw. Installations have been surviving with increasing difficulty with huge tape libraries or walls covered with removable disk packs and boxes of floppies. The inadequacy of the available storage media is compounded by the lack of software to facilitate and administer large on-going data archives. This problem is especially acute in the Unix environment where the various programs which manipulate tapes (presently the most viable backup alternative) are counted among the most arcane and difficult to use. Among the programs available in the Unix environment for the support of magnetic tapes (and thus archives) are the following:

tar Available with both Bell and Berkeley Unix, tar is the program (and format) most commonly used for porting tapes or floppies from one computer to another. Tar can back off individually named files or entire sub-trees. If asked to back off a sub-tree, tar must copy every file therein making requests such as

``backup every file modified in the last 72 hours'' impossible. Tar does not handle multiple versions of the same file appearing on the same tape very well. In fact, if there is more than one file with the same pathname on the tape, only the one appearing last is easily restoreable. Also, there is a rather low limit on the length of a file's pathname (100 characters on Bell System 5 and Berkeley 4.1Bsd). Only some versions of tar know about the finitude of tapes causing multiple reel backups to be messy and not completely portable. In its favor, tar is very easy to use when what is desired is something tar does well. Also, some version of tar is available on all Unix systems.

cpio Available with Bell Unix, cpio is significantly more flexible than tar in that cpio will accept on its standard input a list of path names to operate upon. This is quite powerful when the list is piped to cpio from a program such as find (which can identify all files matching some set of constraints as in the previously cited example where we asked to backup all files modified in the last 72 hours). Cpio prepends a header to each archived file which allows it to be restored directly to its former mode and location. On the negative side the header does not allow for pathnames larger than 128 characters (Bell System 5) and, since it does not know about tapes (only its standard input and standard output) cpio does not handle multiple reel transfers.

dump/restor

Available under Berkeley Unix, dump (and its complement restor) is the standard archive manager under 4.1 and 4.2 Bsd. Dump knows about tape lengths and automatically interrupts its operation when the current tape is exhausted. Dump can be run automatically (provided there is a human around to change tapes) and can be asked to select files based upon how recently they were modified.

Unfortunately, dump does not maintain a database, making it very difficult to identify the appropriate tape to mount when a restoration is desired. Moreover, the data structures used by dump to keep track of what is on a particular dump-set (the set of tapes created by dump during a given execution) often forces the entire dump-set to be read in order to restore a specific file. The restor program extracts files from a dump-set placing them in the current directory. Files, when extracted, are given integer names (not their original pathnames). If it is desired to restore the extracted files to their original names and locations the program dumpdir must be executed, making a second (potentially very long) pass over the dump-set.

The support, both in hardware and software, of archival storage in the Unix environment has to this point been a story of compromise. The strengths are in every respect almost completely negated by the weaknesses. Yet the necessity of information backup remains.

Recently, however, several companies have announced their plans to market disk subsystems based upon optical technology[†]. Laser disk based storage has been field proven in the form of home video disks as well as the ``compact'' audio disk. Laser

[†]In fact, several products are currently on the market. Pioneer and Hitachi to name two firms. IBM is also planning optical storage subsystems.

video disks can contain as much as one hour of video (plus audio) information per side. The ``compact'' audio disk can contain approximately one hour of audio information on a single side of a 5 1/4'' disk. To give some impression of the volume of information this represents consider that each second of audio is broken into approximately 80Kbytes (16 bit digitization at 40KHz sampling rate) of data. One hour of audio will produce approximately 300Mbytes of data (stored on a single side of one 5 1/4'' disk). Video disks may contain 1 to 2 Gigabytes of information per side.

The extremely high storage density is achieved by use of a laser capable of producing an intense beam of light which is focused into a spot one micron in diameter. On writing, the intensity of the beam is sufficient to vaporize an extremely small spot on the surface of the disk. For reading the intensity is not quite as high. Logic levels are detected by examining the reflection of the laser's light, the presence or absence of a burn producing logic 1 or 0.

This would seem to signal the end of compromise with regard to the backup problem.

- ⊕ Extreme low cost (less than a dime per megabyte).
- ⊕ Enormous capacity.
- ⊕ Seeks not much slower than magnetic disk. Data transfers just as fast.
- ⊕ Minimal space requirements (about the same as storing an LP record).
- ⊕ No need for special handling.
- ⊕ Very long life time (nothing ever touches the surface of the disk; also head flying heights are much larger than with magnetic media).
- ⊕ Easily removable and portable.

There is however, a catch. The products available initially will be ``write-once'' optical disks. That is, once a bit has changed state (by being blasted with the laser) there is no way to reset it to its original state[†]. Therefore the data structures used in placing information on the optical disk must be designed carefully. Also, there are a number of considerations which arise purely from the enormous capacity of the medium. For example, a standard Unix file system can support only 16 bits worth of unique file control blocks (inodes). This means that a single standard Unix file system (if placed a 4Gbyte optical disk) would still allow only as many distinct files as perhaps a 1/2Gbyte magnetic disk.

Further complications arise when we consider the intended use of optical disks, that of archival storage. Consider that the standard Unix file system does not allow more than one file in the same directory to have the same name. This would mean that

[†]Erasable optical disks, available in the future, will combine magnetic and optical technologies. Each platter will be surfaced with a magnetic material which when cooled (after a laser blast) in the presence of a magnetic field will, retain one of two magnetic orientations.

in order to keep several back versions of (for example) a C source file, some part of the original information as to the file's name and original location would not be readily apparent. That is, we could either store the source files with the same name in different directories (obscuring their original location) or give the source files different names keeping them in the same directory (but obscuring their original name).

Also, we may make the observation that once a file is archived, the archived copy will not change. The standard Unix disk block allocation scheme does not ensure that blocks associated with a particular file will be located near each other (let alone contiguous). This is a by-product of the way Unix allows additional data blocks to be tagged on the end of existing files. Clearly the ability to dynamically add blocks to a file is needed only when files are allowed to change. Since this is not the case with archival storage we can gain a significant performance increase by storing individual files in contiguous blocks (ie: by executing fewer seeks we more than compensate for the slower seek time offered by optical disks). Additionally, since we are dealing with a write-once medium, any opportunity to do away with a control structure which is subject to frequent change (such as the standard Unix ``free list'') should be taken.

In the following paragraphs we will propose two schemes for integrating write-once storage devices into the Unix environment as well as a scheme for retro-fitting the standard Unix file system with modifications which would allow its use with archives based on erasable optical storage (not to mention existing magnetic disk). In the design of each scheme (for both write-once and reusable optical storage) the following design goals have been upheld:

- ⊕ A minimum amount of change is to be tolerated among the control structures of the archive (this goal is relaxed somewhat in the third scheme since it is intended for use with erasable optical disks).
- ⊕ The archive should be quickly searchable either by self-contained data structures or by a database contained on conventional media. If the index structure is not contained on the optical disk itself then provisions must exist for reconstructing the external database in the event of machine failure.
- ⊕ Information should be stored in contiguous blocks as often as possible.

1. Base-Line Implementation

The base-line alternative is to treat the write-once storage device as a very long tape. That is, store all files in contiguous blocks one after the other with no index structure built into the archive. Each file will be preceded by a header containing enough information to completely restore the file to its original form and location. The header can be adapted from the one used currently by cpio and is shown in Figure 1. The only file on an archive constructed under this scheme which would have the potential to change would be list of ``last-used'' blocks maintained to keep track of the next available location on the disk. The structure of such an archive is given in Figure 2 and explained below:

block 0

Block 0 contains in its first four bytes an integer identifying the disk. The remaining bytes in the block are available for local use.

block 1

Block 1 is the first block in a chain of blocks keeping track of the last used (thus next available) block. If the basic block size of the archive is taken to be 1024 bytes then each pointer block can contain a maximum of 255 last-used/next-available pointers. The 256th (or in general, the last) entry is reserved as a pointer to the next block in the chain. An entry would be made in the last block of the chain after thirty seconds of idle time following a write. The last pointer block is so signified as its 256th entry will be 0. The next available entry in the last pointer block is similarly signified.

block 2 and so on

Block 2 is the header block for the first archived file. The data in the file are not forced to start at the next block boundary. Rather, the data can begin immediately after the end of the header. The information in the file is written in the succeeding blocks contiguously. After the last block of data associated with the file whose header begins in block 2, starts the header of the next file (and so forth).

This scheme, being quite trivial, does not afford rapid indexing through the archive. That task is given to an on-line database system and associated programs (such as query managers, screen handlers etc). Each time an archive disk is loaded, its ID field is checked to see if the on-line database already knows about it. If the disk is unknown, each header contained on the disk can be read and inserted into the database. Since the optical disk is a true random access device and each header contains information suggesting where the next header begins, the files themselves need not be scanned. Updating the database can be done in the same pass as following the list of pointer blocks to discover the last-used/next-available block. Clearly the on-line database is reconstructable in the event of a host failure since reconstruction means only that all disks are ``unknown.''

The on-line database itself can be broken into two parts. A relational database containing the header information of each file and a ``pathname trie'' affording rapid translation to relational keys given a file's full pathname. The relational database would support inverted indices allowing quick access to archived file headers under a broad range of queries. Figure 3 shows the structure of the database suggested here. It should be noted that this scheme is easily simulated upon magnetic tape using the physical EOT (end-of-tape) mark in place of the last-used/next-available list.

The major weakness (or possibly the major strength) of the base-line scheme is the necessity of the on-line database external to the archive. The on-line database might be considered an advantage in that it collects information about many disks in a single well-ordered place. By spanning disk boundaries the on-line database unifies the collection of disks into a coherent archive. The database management system used to administer the on-line database could allow the formulation of queries and relationships not directly expressed in the very simple physical structure of the archive.

The base-line scheme's weaknesses stem mainly from the fact that the on-line database must be constructed and maintained. The integrity of the database must be carefully maintained to ensure that what is on an archive disk is accurately reflected in the database. Also, considering the enormous capacity of optical disks one would expect the size of the on-line database describing the contents of the disks to be large as well.

2. Hybrid Implementation

The on-line database of the base-line scheme is necessary because of the lack of any indexing scheme on the optical disks themselves. The need for the database can be eliminated by introducing an indexing method into the data structure used to manage storage on the disk. This adds a good deal of complexity to the data structures found on the disk and the algorithms which manipulate them since as files are added to the disk, accommodations must be made in the index structure.

The hybrid implementation is so named in that it combines the hierarchical directory structure of the standard Unix file system with the contiguous storage of files found in the base-line implementation. Directories will be allowed to change since they will comprise the index structure of the archive file system. Unlike standard Unix directories which allow entries to be added anywhere there is an available space, an archive directory will only allow additions to be appended to the directory's end. This means that entries once deleted (removing a file from the archive) will not be reused. This is no loss however since the use of the hybrid implementation assumes a write-once disk as the storage medium. Allowing entries to be added only after the end of a directory produces the welcome side effect of sorting the entries in the order of their archiving.

Allowing directories to change reintroduces the need for inode-like structures to keep track of where the individual blocks in a directory are located. Since directories are the **only** files allowed to change, there need not be inodes for non-directories. The standard Unix file system limits the information contained in a directory entry to just a file's name and its inode number. This is reasonable only when it is given that all files have inodes and that all inodes are stored in roughly the same place. Since these assumptions do not hold in the hybrid implementation we expand the contents of a directory entry to include the most often desired fields previously found in a file's inode. Thus, at the expense of an additional 16 bytes per entry we gain a considerable performance advantage over the standard Unix file system when servicing extremely common queries (such as an `ls -l`). The structure of an archive directory entry is given in Figure 4.

We feel that multiple versions of the same directory should not be allowed in the archive. Intuitively, it's not the directory itself that changes over time but rather the files contained in it. We can easily handle the presence of multiple versions of the same file (same pathname) but the presence of multiple versions of the same directory introduces opportunities for terrible performance and ambiguity. Disallowing multiple versions of the same directory is consistent with the view of directories as purely index structures pointing to archived data files.

Since directories will be given inodes (and identified by inumbers) the need arises once more for an ilist (a place for the inodes to live). In the standard Unix file system, the size of the ilist is fixed at the time the file system is created

and cannot be changed. Considering the enormous capacity of optical disks and an unknowable distribution of file sizes it would be unwise to fix the size of the archive ilist to any specific value. With this in mind, we propose a "segmented ilist" structure thereby allowing additional inodes to be created as needed.

The structure of a hybrid-archive implementation is described below:

block 0

Block 0 contains the disk identification code as in the base-line implementation.

block 1

Block 1 contains the first "LU/NA" (last-used/next-available) block as in the base-line implementation.

block 2

Block 2 contains the first "ilist-pointer" block. Each pointer is a four byte value representing (if non-zero) the starting location of 128 contiguous blocks of inodes. Values will be inserted (as required) into this block consecutively starting with the 0th slot up to but not including the last slot. If non-zero, the last slot gives the block number of the next similarly constructed ilist-pointer block.

It should be noted that a single ilist-pointer block can be used to address 522,240 distinct inodes (assumes 1024 byte blocks). Consequently, it is unlikely that a second ilist-pointer block will be required. However, even if several ilist-pointer blocks are required, finding a desired inode is a trivial operation.

block 3 to block 130

These blocks are used to store the first 2048 (with 1024 byte blocks or 1024 with 512 byte blocks) inodes. We emphasize that only directories have inodes. Hybrid-archive inodes need contain only data address pointers since directories are used only as indices into the archive and are not archived themselves. As such, each inode has 12 direct pointers (to data) in addition to the usual single, double, and triple indirect pointers.

Instead of using the remaining four bytes as another direct pointer which would provide only marginal benefit, we have elected to code the inode's name (its "inumber") in each inode. This is of course redundant information since (as in the standard Unix file system) the inode's position in the ilist determines its inumber. But, since the hybrid-archive ilist is segmented and it is possible to destroy an ilist-pointer block, the ilist is easily reconstructable by scanning for 128 blocks worth of sequential integers spaced every 64 bytes. The structure of a hybrid-archive inode is given in figure 5.

block 131

Block 131 contains the first block of directory data for the disk's root directory. The role of the disk's root directory is the same as its counterpart on a standard Unix file system. That is, to provide the root of the directory hierarchy.

block 132 to ...

Block 132 contains an archive header and the start of the data associated with the chronologically first file to be archived on the disk. The header and data blocks of each file are constructed in the same way as in the base-line implementation. We emphasize that all blocks associated with a particular file are stored contiguously.

Figure 6 portrays the overall structure of a hybrid-archive implementation.

The hybrid scheme has many nice features:

- ⊕ It can be implemented on write-once storage technology with minimal overhead. Software can be developed using standard magnetic disks.
- ⊕ The overall structure of the archive resembles closely the general structure of a standard Unix file system preserving knowledge and expertise.
- ⊕ Unix utilities such as ``find,'' ``ls,'' and ``cp'' can easily be modified to operate on hybrid archives. Thus the archive can be searched using familiar tools.
- ⊕ Kernel level support of hybrid archives is a very reasonable proposition.
- ⊕ The archive is robust even in the face of possible disk malfunction. All data structures are reconstructable, meaning that only those individual files which are themselves destroyed in a disk failure will be lost.

The hybrid implementation is presently being implemented by the author. The above mentioned utilities are being modified and new utilities supporting network wide archiving are being developed.

3. Retrofit Implementation

The base-line implementation featured contiguous storage of files on write-once disks and required the use of an external database. The hybrid implementation was also designed for use with write-once disks, allowed files to be stored in contiguous blocks and offered an internal index structure very similar to the standard Unix file system. Following this trend, the ideal implementation would allow archived files to be stored in contiguous blocks while providing a file system structure identical to the standard Unix file system. Unfortunately this cannot be done using write-once optical disks. All hope is not lost, however, as erasable optical disks are only a few years from the market-place.

Using erasable optical disks, the standard Unix file system can be retrofitted to include provisions for archived files stored in contiguous blocks. This is accomplished by allocating another file type flag in the mode field of a standard Unix

inode. If the ``archive-file'' flag is set the only write operation allowed on the file is erasure. If a file is an archive file then its write permission bits are available for other use. We use one of these to signify that the file's data blocks are stored contiguously. We achieve this by noting that a standard Unix file system will initially create a free list of consecutive blocks. When we create an archive file we traverse the free list to its end, verify that there is a sufficient number of consecutive blocks available and allocate them. If there are not enough consecutive blocks available the file can be stored in the standard way. It should be noted that programs exist to reorder a ``lived-in'' free list grouping consecutive blocks together again.

The segmented ilist structure described under the hybrid implementation would be adopted, due (again) to the huge capacity of optical disks and the unknowable distribution of file sizes. We handle having multiple files of the same name by specifying that no more than one non-archive file in a particular directory may have a given name. We stipulate that the Unix system calls ``open'' and ``creat'' only operate on non-archive files allowing them to operate unchanged. We add the system calls ``aropen'' and ``arcreat'' to allow archive files to be created and read. To allow archive files to be referenced from the shell level, we could extend the shell meta-character syntax so that the shell will perform arope's as opposed to open's and allow the user to identify which of several archive files of the same name is desired. For example:

```
cat data.file > A.data.file
```

creates an archive file ``data.file'' while

```
cat data.file >\A.data.file
```

creates an ordinary file named ``A.data.file.''

Admittedly, the retrofit implementation requires additional research before it can be implemented. The idea (of merging the standard Unix file system with an archive file system supporting contiguous block file storage) is a good one and has implications beyond archival storage. For example, video (or audio - or both) information can be created and stored on optical disks as easily as more common data types. Video files can be stored in contiguous blocks allowing smooth play back without having to go outside the Unix file system. The possibilities (in this respect) are staggering.

Conclusion

Optical disks signal the beginning of a new and positive upturn for the ``age of information.'' Video and audio information can be stored side by side with ordinary data files. Available soon, write-once disks introduce subtle problems in the Unix environment. Two schemes have been proposed which would allow the incorporation of write-once disks into the Unix environment with little pain and much benefit. Available in the future, erasable optical disks remove some of the problems associated with write-once disks but problems associated with the Unix operating system's file system structure still remain. A plan for retrofitting the standard Unix file system to solve these problems is outlined. Additional research is warranted into how the Unix file system can be merged completely with huge capacity drives,

presence of archived files and contiguous storage of such files (as well as audio and video files).

Acknowledgements

Special thanks to Alan Weston whose labor produced figures 2,4,5 and 6. Thanks to Jack Heller, an outstanding chairman and friend. To Rick Spanbauer, an outstanding friend (not yet a chairman). And to Pat Madison for reading the draft.

```

struct ar_header {
    short      ar_dev;
    unsigned short ar_ino,
               ar_mode,
               ar_uid,
               ar_gid;
    short      ar_nlink,
               ar_rdev,
               ar_fname_size,
               ar_fmach_size;
    long       ar_size,
               ar_mtime,
               ar_atime;
    char       ar_name[ar_fname_size rounded to word];
    char       ar_machine[ar_fmach_size rounded to word];
};

```

Base-Line Implementation Header Format
Figure 1

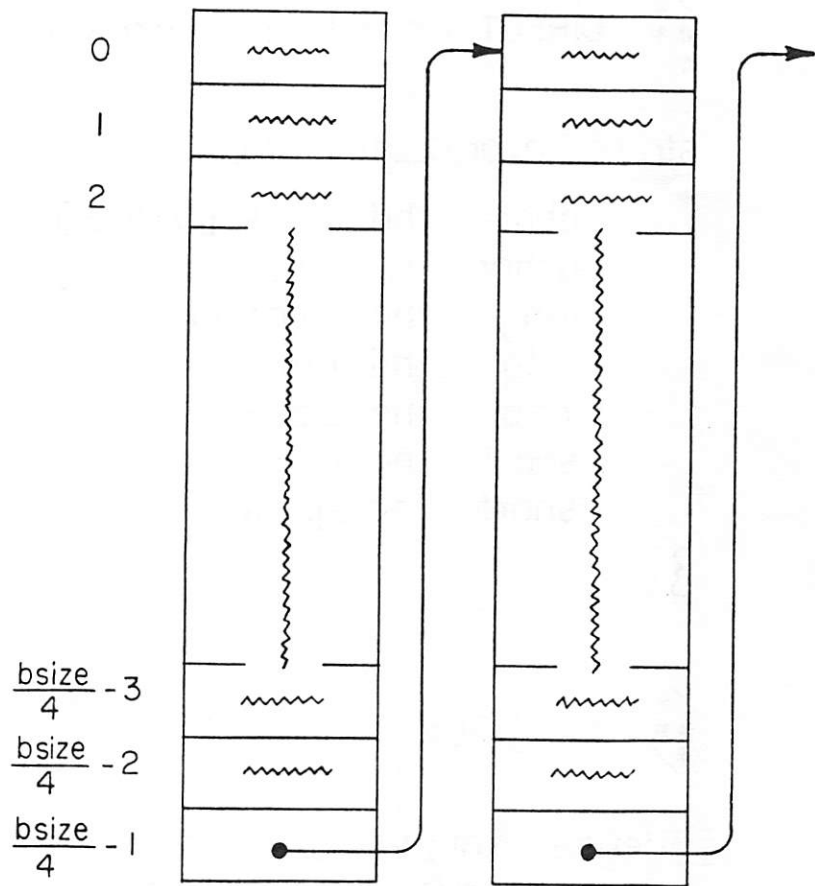
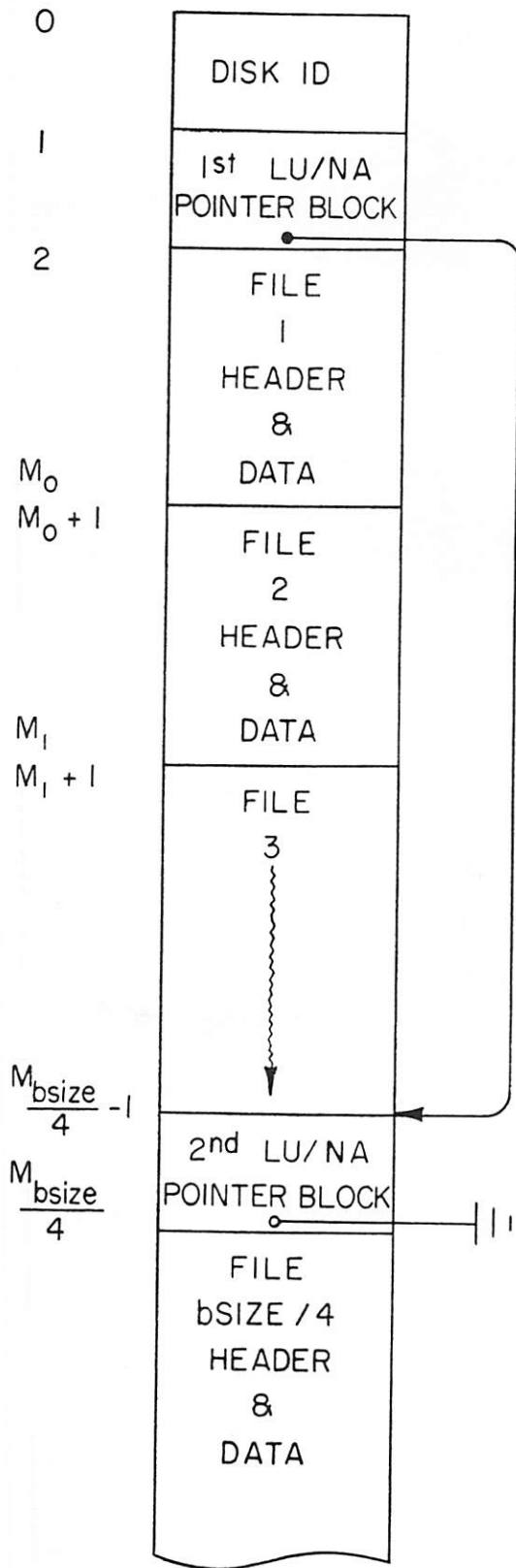
```

struct relation {
    struct key_part {
        long      path_name_id;
        long      disk_id;
        long      start_block;
    } key;
    struct ar_header data;
}

struct pname_trie {
    struct      pname_trie child; /* pathname trie allows */
    struct      pname_trie sibling; /* rapid indexing of re- */
    long        path_name_id; /* lational database by */
    char        character; /* turning pathnames into */
    unsigned char flags; /* partial keys */
}

```

Suggested Database Architecture for Base-Line Implementation
Figure 3



- ONLY THE LU/NA BLOCK WHICH IS CURRENTLY THE TAIL OF THE CHAIN MAY HAVE ZERO'ed ENTRIES.
- INITIALLY, ALL ENTRIES IN THE TAIL BLOCK ARE ZERO.
- ENTRIES ARE USED SEQUENTIALLY STARTING WITH THE 0th ENTRY.
- THE $\frac{b_{size}}{4} - 1$ ENTRY, IF NON-ZERO POINTS TO THE NEXT LU/NA BLOCK.

OVERALL STRUCTURE OF A
BASE-LINE ARCHIVE

DETAIL OF LU/NA POINTER BLOCKS

FIGURE-2

```

/*
** DIRECTORY ENTRY - HYBRID IMPLEMENTATION
*/

```

```

struct hybrid_dir {
    char    hd_name [DIRSZ] ;
    ushort  hd_flags;
    long    hd_location;
    long    hd_mtime ;
    long    hd_size ;
    short   hd_owner ;
    short   hd_group ;
};

```

```

/*
** hd_flags DEFINITIONS
*/

```

```

#define DELETED_ENTRY      0xFFFF
#define AVAILABLE_ENTRY    0x0000
#define IS_DIRECTORY      0x0001
#define IS_EXECUTABLE      0x0002
#define RESERVED          0xF000    /* MASK */

```

```

/*
** HYBRID INODE
*/

```

```

struct hybrid_inode {
    long    ino ;
    long    direct [12] ;
    long    sindir ;
    long    dindir ;
    long    tindir ;
};

```

FIGURE-4 & 5

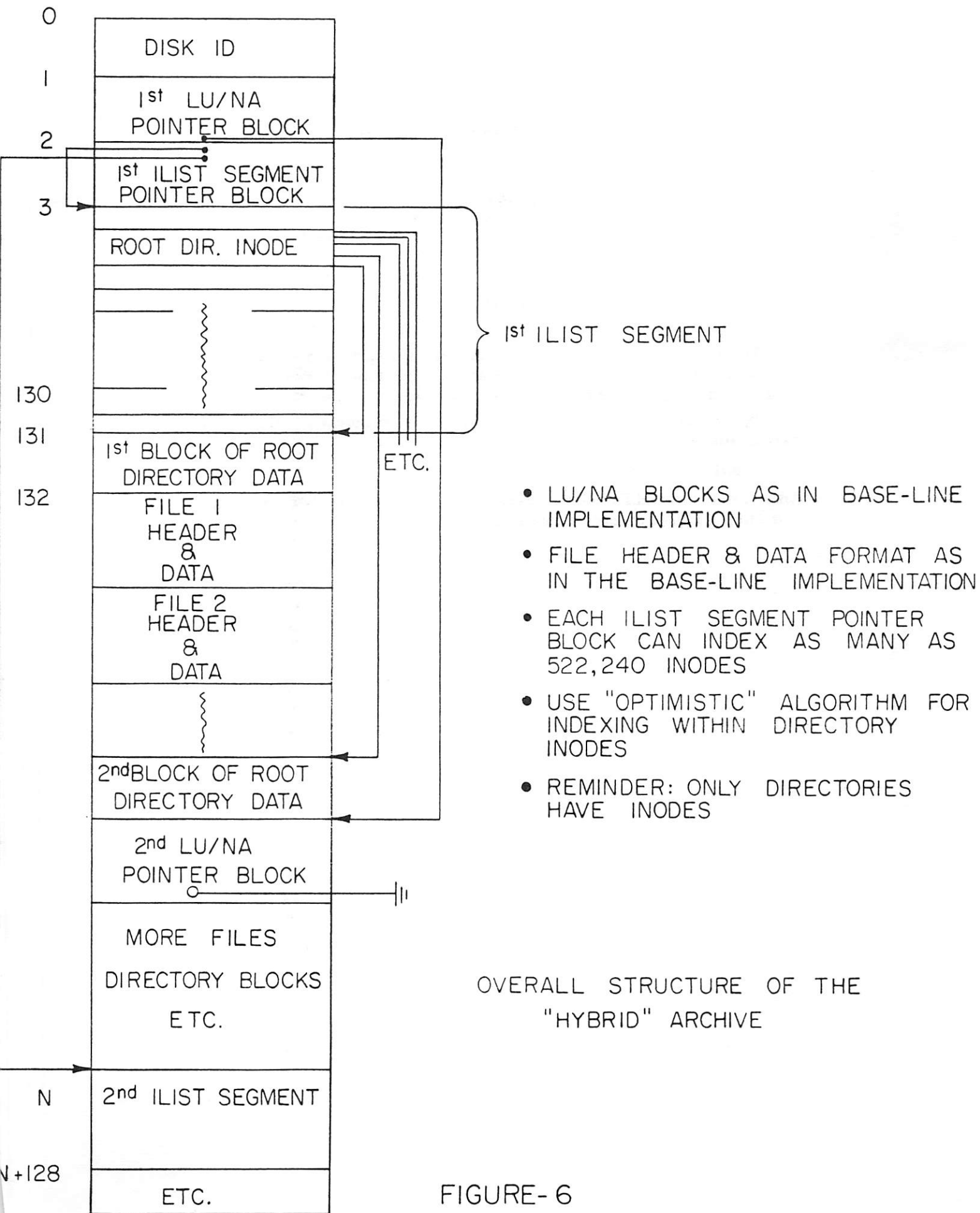


FIGURE- 6

Automatic Software Distribution

Andrew Koenig

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The author of a popular program frequently has trouble ensuring that each user has a copy of the current version. When there are many users, the problem is made even tougher by the need to maintain data integrity and security.

We discuss some tools we have built to help solve this problem. These tools make it possible to execute:

`ship /bin/who`

After one has supplied the correct password, this will eventually cause new copies of the *who* command to appear on many remote machines.

Automatic Software Distribution

Andrew Koenig

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

Our lab operates a number of small computers, each running the UNIX[†] operating system. Because these machines share a common hardware architecture, we try to keep only a single copy of the system's source programs and replicate the object programs as necessary.

Many of our people are actively writing new software. When a new program is ready, the author typically puts a copy of the program on one (or a few) of our machines and tells people about it. Potential users who hear about the program may copy it onto their own machines.

If the program is popular, copies eventually migrate to several machines. Since the program has probably changed during this period, each machine might have a different version. An author who learns of a bug from a user may have trouble deciding whether that bug still exists or whether the user is running an obsolete version of the program.

For example, Mike Lesk has mentioned that two thirds of the bug reports he gets on one popular program he wrote refer to things he has already fixed in the latest version.

Why does this happen? The main reason is that it is too much work for an author to keep copies of his program up to date for all users. If a user is reasonably satisfied with the behavior of a program, there is usually little incentive to install an update. Even if a user is dissatisfied, that user might have to go to the system administrator to get a new version installed.

It would be nice if all users of a program always had the latest version. To make this more firm, though, we must answer several hard questions: What does it mean to have the latest version of a program? How does one go about installing a new version? What about data integrity and security?

What's in a version?

Imagine that the author of a widely used program, such as the *ls* command, could somehow send new versions of that program to all its users. To send out a new version, the author might send three files: */bin/ls*, the executable object program, */usr/man/man1/ls.1*, the text of the reference manual page that describes the program, and */usr/src/cmd/ls.c*, the source program. In the minimal case, one need only send */bin/ls*. If the receiving machines have different hardware, it may be useless to send */bin/ls* directly. Instead, the receiving machine must be told to create a new */bin/ls* from the source code.

We learn several things from this simple example. First, sending a new version of even a straightforward program may involve updating several files on each receiving machine. Second, we may need to send different things to different machines. Third, installing a new version of a program may involve more than just copying data (such as system-dependent initialization).

[†] UNIX is a Trademark of AT&T Bell Laboratories.

What granularity?

Should it be possible to send only part of a file? Perhaps. For example, a distribution mechanism might be used to effect user enrollment on a community of machines.

Unfortunately, partial file updating raises two questions. First, just what updating operations should be supported? Second, how do we ensure that the file was correct before modification?

Replacing only entire files avoids both problems. Moreover, if it is possible to execute a program on the receiving machine as part of installation, it is unnecessary to provide another mechanism for partial updates.

Structure

Part 1 of this paper describes *mkpkg* and *inspkg*, tools for creating and installing packages. Part 2 describes *seal* and *unseal*, which make it difficult for a package to be modified in transit without detection. Part 3 describes a simple "gatekeeper" that permits *inspkg* to be executed with the appropriate privileges while limiting the risk of unwanted execution. Part 4 describes a simple shell script that puts all these pieces together into a system. Part 5 discusses the current state of that system and hints at future directions.

1. Putting data into packages

Because updating a piece of software often requires changing several files together, it is important to have a data structure that contains all the information in a collection of files. We call this structure a *package*, and manipulate packages with *mkpkg* and *inspkg*. *Mkpkg* gathers a set of files into a package, and *inspkg* uses that package to re-create a copy of the files. For example,

```
mkpkg a b c >foo
```

collects the contents of files *a*, *b*, and *c* into a single file named *foo*, and

```
inspkg foo
```

unpacks *foo* and re-creates the original files.

A package contains a copy of the contents of each file stored in it, as well as that file's full pathname, its owner, group, mode, and reference and modification times. As many of these as possible are restored when the file is re-created. If *mkpkg* is given the name of a directory, all files in that directory are (recursively) included in the package.

Re-creating a file generally involves deleting it first, in case a file with the same name already existed. This is true even if the file is a directory: all files in that directory are deleted, and the directory itself is then removed. To guard against possible inadvertent destruction of large amounts of data, *inspkg* has a backup option, which causes it to create a package of everything destroyed during installation. The backup package can be installed by *inspkg* in the usual way.

When an argument to *mkpkg* names a non-existent file, the resulting package includes an instruction to delete that file.

Sometimes it is useful to install files at a different absolute pathname than implied in the package. For instance, programs in */usr/bin* on one machine might be more appropriate for */usr/lbin* on another. Similarly, it can be useful when making a package to pretend that a file is somewhere other than its true location. To these ends, *mkpkg* and *inspkg* allow a list of pairs of pathnames to be given. Whenever a file is found whose name matches the first element of a pair, the matching prefix is replaced by the second element of the pair.

Finally, it is possible to "pretend" to install a package: a backup is made and the usual checks are done, but no files are actually changed. This allows a user to see what files are affected by a package without doing any damage.

The package data structure contains only printable characters. Thus, a package containing only entirely printable files is itself entirely printable. This is intended to make debugging easier, at the cost of a good deal of complexity in the programs themselves.

What's in a file?

When sending a file, it is often important to preserve more than just the file's name and contents. For instance, some commands (such as *mail* and *uucp*) must be owned by particular users, and modification times may be important for libraries and collections of files accessed by the *make* program. When preserving the owner of a file, it is important to use the character representation of the owner, rather than the absolute numeric *uid*: common practice is for the same individual to have different *uids* on different machines, but the same login name. The protection bits are also important. Special files do not have contents as such, but the major and minor device numbers are important.

What's in a package?

A package is actually an archive (as if generated by the *ar* command), with one component for each file it "contains," and an extra component (named *Instructions*) that tells *inspkg* how to install it. Choosing this format is actually responsible for a good deal of the complexity in *inspkg*, but it makes it possible to use the *ar* command to break open a package in an emergency.

The *Instructions* component contains a sequence of instructions to *inspkg*, each one on a single line. The first character of an instruction identifies the type of instruction; the format of the rest of the instruction is type-dependent. For example, an instruction that describes a file has a type letter of *f*, followed by the name of the archive component that contains the file, the owner's user and group id, and the full pathname of the destination. File instructions do not include date or mode, as this information already appears in the archive header. Other instructions can cause *inspkg* to make directories or links, or delete files.

Inspkg insists that the archive components be in the same sequence as the corresponding instructions.

Naming conventions

All files named in a package are remembered by their full pathnames. *Mkpkg* helps in this by putting the name of the current directory in front of any relative pathnames. Thus *inspkg* need deal only with full pathnames.

Sometimes it is useful to install a file in a directory with a different name than the one that originally contained it. Both *mkpkg* and *inspkg* therefore allow a primitive name replacement. For example, suppose we want to create a package containing a new command that is stored in */usr/bin/lcl/newcmd* on our machine. We may want the package to indicate that this file should be installed in */usr/bin/newcmd*. This is done as follows:

```
mkpkg -D/usr/bin/lcl=/usr/bin /usr/bin/lcl/newcmd >newpkg
```

Similarly, the recipient of this package may want to install the command in */usr/nbin*. The way to do this is:

```
inspkg -D/usr/bin=/usr/nbin newpkg
```

As with all pathnames given to *mkpkg* and *inspkg*, relative names are made absolute.

Once a *-D* option has caused a name to be changed, that name is not touched further. Thus

```
mkpkg -Dx=y -Dy=x x y >z
```

will put a package in *z* that contains files *x* and *y*, and will pretend that the names have been interchanged.

The *-b* option requests a backup package:

```
inspkg -b newpkg >oldpkg
```

Oldpkg will contain everything that was deleted during installation of *newpkg*.

Installation note

Some UNIX systems do not allow a file to be modified while a process is executing it. For this reason, files are installed by first deleting any old file that might exist and then re-creating a new file with the right characteristics. If *inspkg* cannot delete the file, it tries to overwrite it anyway, just in case it succeeds.

Once the file has been created, *inspkg* tries to set its owner and group. On some systems, this can only be done by the super-user.

2. Tamper-resistant packages

Once upon a time, some people were trying to break into a computer system. They tried everything they could imagine, but were unable to get in. Finally, they hit on a clever idea. They made their own copy of the system and changed that copy to make it easy to break in. They then put the changes on a reel of tape, put the tape in a box with the name of the operating system's manufacturer on its return address label, and mailed it to the system administrator. The administrator, a trusting soul, looked at the box and said: "Aha, another system update. Let's install it!"

Valuable objects are generally shipped in sealed packages. Such packages usually serve two purposes. First, they make the contents easy to handle safely. They do this, in part, by enclosing the objects in a box whose shape makes it easy to transport. Second, they make it difficult for anyone to tamper with the objects without being detected. It is extremely difficult to prevent a box from vanishing altogether, but that is usually not the sender's motivation. Rather, the sender wants to ensure that any damage done in transit is detected quickly, so that claims can be pursued against the shipping company.

Suppose software updates are installed automatically. How can the receiving machine ensure the identity of the sender? It is clearly insufficient for the sending machine to announce its name: that, by itself, is too easily imitated. It would be nice to be able to defend against a rogue superuser on the sending machine, but that is probably too much to ask: the programs that send things out are subject to corruption, too. The most we can hope for is to require the author to supply a password when sending things: it is best that the password not be stored on the sending machine at all.

Much can go wrong when sending data to another site. Some transmission programs will only accept printable characters, grouped into reasonably short lines. These programs may quietly garble data that do not fit their standards. Data can also be corrupted by hardware errors or line noise. Bugs in transmission software can cause data to be corrupted or lost. Data, particularly if of economic importance, can be altered by direct human intervention. In general, it is extremely difficult to prevent data from being altered, but it is important to know about it when it happens.

A similar problem arises when trying to send valuable physical objects through the mail. Things can be protected against physical damage by cushioned packaging, and packages can be sealed to protect against pilferage, but it is impossible to prevent an entire package from disappearing.

The outer box on a package thus serves two purposes: it makes it more convenient for the shipper to transport the contents (by enclosing it in a regular shape), and it makes it hard for anyone to affect the contents without leaving externally visible signs.

We accomplish these purposes with two commands called *seal* and *unseal*. After executing

```
seal a b c >x
```

the file *x* will represent the concatenation of *a*, *b*, and *c*. This representation contains only printable characters, divided into lines of reasonable length. Executing

```
unseal x >y
```

will unscramble this representation, and make *y* into a copy of the concatenation of *a*, *b*, and *c*.

In effect, these commands are a roundabout way of saying


```
cat a b c >y
```

The difference is that *x*, the intermediate form, is protected against inadvertent damage in transit. If more security is important, one can say

```
seal -k a b c >x
```

Seal will now ask for a key. To unseal *x*, done by saying

```
unseal -k x >y
```

the same key must be supplied.

Note that *seal* does not attempt to encrypt the data being sent, or to prevent the data from being modified. Instead, it makes it hard to modify data *without detection*.

If *unseal* finds something wrong, it writes nothing at all on the standard output (and complains appropriately on the standard error file). This means that the output from *unseal* can be safely piped into other programs without further testing. In particular, consider the pipeline

```
mkpkg file ... | seal | channel | unseal | inspkg
```

Here, *channel* might represent the process of sending data from one machine to another, and might therefore be somewhat unreliable. *Unseal* ensures that if it finds an error in its input, *inspkg* will be given a null input file.

How it works

Sealing a package has two purposes: allowing it to be sent easily and protecting its contents against change.

Easy transportation is assured by using an escape scheme to encode the bits that make up unprintable characters. Files with too many unprintables are encoded a word at a time, to make sure that the encoded file is not too much larger than the original. The method is more complicated than it needs to be, but has the advantage that text files are often changed very little during encoding.

Seal tries to preserve data integrity by putting the length and a checksum of the data at the end of the sealed file. *Unseal* expects these to match exactly. If a key is given, *seal* encrypts the data before taking the checksum, but does not send the ciphertext. *Unseal* duplicates the process: giving the wrong key will result in a checksum mismatch.

It is important to realize that merely encrypting the checksum isn't good enough. Although this would make it hard for an interloper to change the *checksum*, it is open to attack by someone who can figure out how to make dangerous changes that leave the checksum invariant. This would be easy if the data being sent were, for example, a source program with comments: changes to the program text (not changing the length) could be balanced by appropriate changes to an arbitrarily chosen comment.

Options

We have already encountered the *-k* option, causing both *seal* and *unseal* to demand a key. Sometimes it is useful for the key to come from somewhere other than the terminal. Thus both *seal* and *unseal* also provide a *-K* option. This is followed by the name of a file that contains the key. There is no way to specify the key directly on the command line, because it is too easy to discover such a key by using the *ps* command.

3. A simple gatekeeper

We have now seen how to transport a collection of files from one machine to another. First, put them in a sealed package:

```
mkpkg file ... | seal -k > package
```

Second, transport that package to the destination machine. The transportation method need not be highly secure. Third, unseal the package and install the contents:

```
unseal -k package | inspkg
```

Because *unseal* only produces output when everything works flawlessly, it is even safe to make this process more automatic:

```
unseal -K /etc/secretkey package | inspkg
```

provided that the file with the key can be kept secret. Given this, and given the fact that these files may be received by daemons with no special privileges, it seems useful to have a program that will automatically install files sealed with the proper key. This program is called *asdrvc*.

We take the view that each individual who uses this tool kit to update programs on our machine will have a separate password, and that an individual is uniquely identified by a machine name and a login id. We use the file system to do our mapping by assuming that the key is in a file named */etc/asd/keys/machine/user*.

Asdrvc has a simple job. Suppose a file named *package* is intended to contain a sealed package created by *research!ark*. We merely tell *asdrvc* the purported creator of this package:

```
asdrvc research ark < package
```

Asdrvc can safely execute the following pipeline:

```
unseal -K /etc/asd/keys/research/ark | inspkg
```

because if the package is not what it seems, *unseal* will give an empty input to *inspkg*, and nothing will be hurt. In fact, we add an extra measure of security by executing, in effect:

```
(unseal -K /etc/asd/keys/research | inspkg) 2>&1 | mail research!ark
```

In other words, the standard output of *inspkg* and the standard error files of *unseal* and *inspkg* are merged and sent as mail to *research!ark*.

Installation permissions

Asdrvc is actually a little more complicated than this. It runs as super-user, but consults a file to find out what user to impersonate. The name of this file again depends on the claimed creator of the package; in the example given above it would be */etc/asd/perm/research/ark*.

This file must exist, or *asdrvc* fails. If it is empty, *asdrvc* runs as the super-user. Otherwise, it contains the name of a user and group, separated by white space. *Asdrvc* looks up these items in the password file and impersonates that user and that group. It is therefore possible to allow particular authors to modify only files writable by particular users or groups.

Once *asdrvc* is installed on a machine and */etc/asd/keys* and */etc/asd/perm* have appropriate permissions and contents, installing a new version is just a matter of arranging for *asdrvc* to be executed with the proper arguments and standard input. Suppose, for instance, that *mach1!user* wants to put a new version of the *who* command on *mach2*. It could be done this way:

```
mkpkg /bin/who | seal -k | uux - 'mach2!asdrvc mach1 user'
```

The call to *uux* eventually causes *asdrvc* to be executed on *mach2* with arguments *mach1* and *user*, and a standard input that is probably identical to the output from *seal*. If something goes wrong, *mach1!user* will be told about it.

4. Putting it all together

Here is a simple way to use these tools to distribute software. Our machines have something very similar to this in */usr/bin/ship*:

```
T=/usr/tmp/asd$$
trap 'rm -f $T.[12]; exit 0' 1 2 3
if mkpkg $* >$T.1
then
    seal -k $T.1 >$T.2
    rm -f $T.1
    (trap '' 1
    for i in ${dest:='comm -23 /etc/asd/machines /etc/whoami'}
    do
        uux - "$i:asdrvc 'cat /etc/whoami' 'getuid'" <$T.2
    done
    rm -f $T.2)&
else
    rm -f $T.[12]
fi
trap 1 2 3
```

The first two lines invent names for temporary files and arrange that those files will disappear if the script is interrupted.

The third line begins the real work. It tries to make a package of all the things to be sent and put it in a temporary file. If unsuccessful, it removes the temporary file and exits.

Suppose *mkpkg* succeeds. The next line seals the package (asking for a key), and puts the sealed package in a second temporary file. We now enter a loop that does one iteration for each destination machine. The destinations' names are stored in */etc/asd/machines* and the current machine's name is in */etc/whoami*. Thus, the expression involving *comm* removes the source machine from the destination list, thus allowing the list to be identical on all machines. This expression is used to initialize the variable *dest* if it is not already set.

The next line invokes *uux* for each destination machine. As we have already seen, this will eventually cause *asdrvc* to validate and install the package on each destination. Since all these invocations of *uux* take some time, we do them in the background.

Finally, we clean up the temporary files and exit.

How to use it

To propagate a file around our machines, it is merely necessary to say

```
ship file
```

and supply the proper password. To send a file to specific machines, say

```
dest='machine1 machine2 ...' ship file
```

In order for all this work, each receiving machine needs two files. */etc/asd/keys/machine/user* contains the key for files sent from *machine!user*, and */etc/asd/perm/machine/user* contains the user and group ID under which things sent by *machine!user* will be installed. The files and directories in */etc/asd/perm* should generally be owned by the super-user, with write permission for the owner only and global read permission. Each file in */etc/asd/keys* should be readable only by the user named in the corresponding file in */etc/asd/perm*. We keep the directory */etc/asd/keys* and its sub-directories owned by the super-user, with execute permission only for others. This is mostly out of a sense of general paranoia: it is probably enough to protect the original files.

5. Discussion

About a dozen people in our lab are presently using these tools to install software on 25 machines. We have given a copy of the tools to one other organization; they are now using them to send things around to five machines.

The current design has a few problems. First, its security depends on being able to keep the keyfiles secret. This is a bad idea for the long run — as the number of machines using a particular key grows, so does the chance of someone's discovering a key through administrative carelessness. We believe that the right way to solve this problem is ultimately by using public-key encryption, but this is somewhat of a nuisance to implement and may run into performance problems. An intermediate measure might be to store a different key on each receiving machine. This key could be systematically derived from the author's master key and the name of the receiving machine, perhaps by applying an effectively irreversible hash function to the concatenation of the two. Such a precaution would make each password hold the key to only a single machine, but it would still be important to keep the passwords secret.

Another potential security problem is that a package once created and sealed with the proper key, is quite a potent item. For example, an interloper who kept copies of old packages could use them to re-install old versions of programs. This would normally do little harm, but someone with an interest in keeping a particular bug around could do so for some time. This can eventually be avoided by making the gatekeeper somewhat smarter. It could, for instance, reject any attempt to install a package that is older than what it is replacing.

It is tough to decide whether *asdrvc* should run *inspkg* with the *-v* option or not. If the option is used, running *ship* eventually results in a flood of mail messages, often too many to be worth sifting through by hand. On the other hand, without *-v*, it is hard to know when the new version of something has finally been installed. Sometimes packages get lost in transit, and a return receipt is a good way of guarding against that.

The way to handle this in the long run is probably to institute some kind of automatic acknowledgments. For instance, instead of running *mail* on the receiving machine, *asdrvc* could use *uux* to run some acknowledgment program on the sending machine. *Ship* could put the outgoing package on a list of unacknowledged items, and the acknowledgment program could remove them from this list. More complex schemes can also be imagined.

Finally, attempting to distribute to too many machines will ultimately cause severe performance problems. We feel that such problems should be addressed as part of the transmission medium, not as part of the distribution scheme. By assuming that the transmission medium might be insecure, we are in a better position to take advantage of new media as they come along. For example, it should not be difficult to use the USENET software to distribute files to many hundreds of machines if there is a demand for it.

Acknowledgment

Mike Lesk, who wrote the first version of *uucp*, first suggested that *uucp* might eventually become a way to help keep programs up to date on a multiplicity of machines. My work is inspired by that suggestion.

Att: manual pages

NAME

mkpkg, inspkg — make and install packages

SYNOPSIS

mkpkg [options] files

inspkg [options] files

DESCRIPTION

These programs allow arbitrary files to be collected into a structure called a *package*, and for the objects constituting a package to be broken apart and installed, typically on a different system.

Mkpkg collects the named files into a package, which is written on the standard output. If a file does not exist, the package will include instructions to *delete* that file during installation. *Inspkg* installs the named packages. Installation should normally be done only by the owner of the files being installed or by the super-user. *Inspkg* itself has no special privileges.

The information contained in a package includes the full pathnames, owners, groups, file modes, and modification dates of objects to be installed, and a list of objects to be deleted during installation. Owners and groups are stored as their character representations, to cater to the possibility of different systems using different numeric codes for a single owner or group.

Options modify the details of the programs' actions:

- v Verbose mode.
- n *Inspkg* will skip the actual installation, but will verify its input package(s) and produce a backup if requested.
- b Backup mode. *Inspkg* will write a package on the standard output that contains everything that was destroyed (or, if the -n option is used, would have been destroyed) during installation.
- Dpath1=path2
 Pretend that any pathname beginning with *path1* really begins with *path2*. Both *mkpkg* and *inspkg* allow this option.

NOTES

A package is really an archive with an extra component named **Instructions**. This component contains additional information needed to complete the installation process. The exact format of this additional information is changing, but it is intended to be human-readable. If all files in a package contain only printable characters, the package will contain only printable characters.

NAME

seal, *unseal* — prepare data for shipment

SYNOPSIS

seal [*-k*] [*-K* *keyfile*] [*file* ...]

unseal [*-k*] [*-K* *keyfile*] [*file* ...]

DESCRIPTION

The output of *seal* is a file containing the same information as the concatenation of all its input files, in a form suitable for shipment by *mail*(1). Its output contains only printable ASCII characters, with no lines longer than 128 characters, and with no lines consisting only of a single period. If no input *files* are specified, *seal* reads its standard input.

Unseal reads files produced by *seal* and produces the original file contents as its output. If *unseal* detects something wrong, it complains on the standard error file and produces no output. Just about the only way that *unseal* can produce an error message together with output is if some kind of hardware error (or the file system runs out of space) while writing the final output.

In addition to the information contained in its input, a sealed file includes a checksum to guard against inadvertent modification. If the *-k* option is specified, the program prompts for a key and uses that key to encrypt the data before calculating the checksum. This enables it to guard against deliberate modification as well.

If the *-K* option is specified, the program uses the first line of the given *keyfile* to supply the key. Subsequent lines are ignored.

4bsd UNIX TCP/IP and VMS DECNET: Experience in Negotiating a Peaceful Coexistence

(extended abstract)

Van Jacobson

Craig Leres

Joseph Sventek

Wayne Graves

Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

Like many organizations, Lawrence Berkeley Laboratory has two large, incompatible, local networks: A 150 node DECNET¹ mostly comprised of Vaxes running VMS; and a collection of Vaxes & 68000s running 4.2BSD, speaking to each other and to the Internet via the UNIX² 4BSD TCP/IP software. We have been working since early 1981 to resolve the incompatibilities and somehow interconnect the VMS and UNIX networks. Recently, we succeeded.

There are two ways to go about the interconnection: either teach BSD UNIX to speak DECNET or teach VMS to speak TCP/IP. Adding DECNET to 4BSD would seem like the simplest approach. 4BSD was designed to support multiple protocols and all the hooks exist to add a "DECNET" protocol suite. Much of the NSP (logical link) and routing layers of Phase-III DECNET have already been done for 4.1BSD by Bill Shannon of SUN Microsystems (formerly of DEC).

Although we investigated "DECNET under UNIX" (and are still planning to implement portions of the transport protocols under 4.2BSD), it seemed to be the wrong choice for a number of reasons:

- The transport protocols are of limited utility to the user community; the user interest lies in higher level *remote terminal* and

file transfer protocols. Unfortunately, the DECNET versions of these protocols are not public and their implementation appears to be non-trivial. (E.g., the VMS implementation of the DECNET file transfer protocol is 50,000 lines of assembler. We were unable to generate much enthusiasm for (a) decoding it, or (b) re-implementing it under UNIX).

- DECNET has limited capability compared to TCP/IP; It lacks a user-level datagram facility, has a small (8 or 16 bit) host address space, lacks notions of multiple networks and gateways, and has a number of configuration parameters that must be known and obeyed globally.³ Even if we solved the higher level protocol issues,⁴ we foresaw major difficulty mapping such things as the IP/ICMP routing, UDP and EGP onto DECNET. Without them, the VMS hosts would appear quite "brain damaged" to the Internet.
- DECNET is a one-vendor 'standard.' Much of the interest in TCP/IP was generated by experimenters who wished to connect low-cost 68000 and 16032 style super-micros to their VMS Vaxes. Needless to say, none of these micros support DECNET. We would be faced with either implementing our UNIX DECNET support on those micros or interposing a UNIX-to-VMS gateway between the experimenter's Vax and micros.

Given these objections, we invested most of our energy in a TCP/IP suite for VMS. We know of two VMS TCP/IP implementations: the *Access* package from Compion and the *Eunice TCP/IP* package from Woolongong. We have run both. The Compion package is a 'clean' VMS implementation but its performance seems poor (about 10 times slower than the Eunice package on a simple

This work was supported, in part, by the United States Department of Energy under Contract Number DE-AC03-76SF00098.

¹VAX, VMS and DECNET are trademarks of Digital Equipment Corp.

²UNIX is a trademark of Bell Labs.

³E.g., *maximum packet size* since there is no provision in the protocols for packet fragmentation and re-assembly.

⁴For example, we considered implementing **Telnet** and **Ftp** using DECNET circuits rather than TCP circuits. The work involved in moving the UNIX User Telnet and Ftp to VMS is small. The work involved in Server Telnet and Ftp is larger but doable.

file transfer), it uses many processes (VMS performance degrades rapidly if it is asked to run many processes) and it supports only the standard ARPANET servers, not Berkeley extensions like *rlogin*.

The Woolongong package was done by David Kashtan of SRI. It is a complete port of 4.1c BSD networking code (i.e., all 15,000 lines of code in *net*, *netinet*, and *vaxif* were moved *in toto* and almost intact — roughly 84 lines were changed). A substantial amount of code was added to simulate the UNIX *socket* user interface under VMS, to simulate the UNIX kernel *sleep* and *wakeup* functions and to gracefully share resources like Unibus Adapters and the System Page Table between VMS and the Network code. The result performed well, had minimal system impact (one additional process) and included all the 4BSD network features (at least, the features that existed as of 4.1c).

LBL was an early test site for SRI's Eunice TCP/IP. We were pleased with its capabilities and performance, particularly when we found that we could move user-level code (e.g., *User Ftp* or *rdist*) from UNIX to Eunice/VMS unchanged.⁵ We were left with two problems: the prospect of having to duplicate our DECNET cable plant and the burden of 4.1c code maintenance on our 4.2 development activities.

The Eunice network software runs the UNIX device drivers. Either Eunice or DECNET can use a device, but they can't share or multiplex it. Since most of our DECNET links are point-to-point DMR-11s, we were faced with the prospect of installing duplicate links for every VMS host that wanted to run TCP/IP. Rather than invest another million dollars in cables, we took a look at what CSNET had done to ship IP packets over Telenet X.25 links⁶ to see if we could

do the same thing using DECNET circuits. Fortunately, the BSD network architecture makes this easy: We added a tiny (100 line) UNIX device driver to put IP output packets in the 'raw' input queue and 'raw' output packets on the IP input queue. We then wrote a relatively small (1000 line) process that reads the IP packets from a 'raw' socket and writes them to DECNET and/or writes packets from DECNET to the 'raw' socket. We then spent two weeks staring at microfiche to decode DECNET, fixing small bugs in the UNIX 'raw packet' and Eunice network initialization code, fixing large bugs in our DECNET bridging code, and crashing our Vax. In late March, we managed to telnet from a DECNET-only host to MIT-MC, with each packet traveling through 4 DECNET nodes and at least 6 Internet gateways. The software has been running continuously on the four Vaxes of our research network since that time, with no network-related crashes and no problems. Any host on our DECNET can now have complete access to the Internet without adding new wires, devices or drivers and without sacrificing any DECNET capability.

The remaining problem is that of the 4.1c network code. There is an enormous user base for the 4.2 code, its bugs are being found (and, occasionally, fixed) and several important enhancements are being made to it (subnet routing, EGP and a name server, for example). As of this writing, we've moved about half of the 4.2 code to Eunice, replacing the 4.1c code. The major remaining piece is the 4.2 routing code, including the new support for subnet routing, which we expect to have going in early June.⁷

We are still in the process of system performance investigation but have some preliminary results. It probably comes as no surprise (at least, to a UNIX audience) that we find 4BSD TCP/IP has substantially better performance than VMS DECNET. In our tests to date, the UNIX throughput has been higher and CPU utilization lower than VMS. The Eunice TCP/IP performance is not as good the UNIX per-

⁵UNIX network server code presents more of a problem: Because of VMS's distaste for processes, one generally rewrites a UNIX server to multiplex via VMS ASTs instead of letting it run in the sensible UNIX style of forking off a child to handle each new connection.

⁶*CSNET Protocol Software: The IP to X.25 Interface*, Douglas Comer and John Korb, Purdue University, Proceedings of the SIGCOMM '83 Symposium on Communications Architectures and Protocols,

University of Texas at Austin, March, 1983, pg.154.

⁷Dave Kashtan is reportedly also converting to the 4.2 code in the process of moving the network from VMS V.3 to VMS V.4. We hope to combine our efforts.

formance, probably owing to the high 'cost' of a VMS QIO (we are currently developing a kernel profiler for VMS to gain a better understanding of where the network, and system, spend their time.)⁸ We were suprised to find that, in most cases, the Eunice TCP/IP outperforms VMS DECNET in both throughput and CPU utilization. The DECNET "bridging" software, of course, has to traverse both the Eunice network code and the DECNET code. At best, its throughput can be

$$X_{dbridge} = \frac{X_{eunice} X_{decnet}}{X_{eunice} + X_{decnet}}$$

We find the DBRIDGE throughput is exactly the above less the overhead of a VMS context switch (1ms every 4 packets). That is, the trip through the TCP/IP 'raw' protocol code has a neglible effect on the performance.

The final paper was not available in time to be published with the proceedings. It will be available from the Technical Information Department of Lawrence Berkeley Laboratory. It will contain more detail on the network implementation, performance experiments and user experience.

⁸The user level interface ("socket system calls") to the Eunice network is via QIOs to a pseudo-device handler that contains all the network code.

Experiences with a Large Mixed-Language System Running Under the UNIX Operating System

Richard A. Becker

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

S is a system for data analysis, graphics, and data management that runs under a number of versions of the UNIX[†] operating system on a variety of hardware. S is large and is written primarily in FORTRAN because of the general availability of FORTRAN routines for performing statistical computations. However, a set of primitive operations for the system is implemented in C.

With S, we have used a form of "lazy portability"; the code is written in portable languages (FORTRAN and C) and the many difficult issues encountered in interactive computing such as handling of user-interrupts, graphics device drivers, dynamic process creation, file access, etc. are handled by UNIX system calls. Thus, as UNIX is ported to other hardware, S goes along with it.

We have recently gained experience with this form of portability. The new UNIX-based workstations are a desirable environment for S because of their low price, good graphics (bitmap, dynamic), and responsiveness; thus we have attempted to make S run on several of these workstations. We now have experimented with the following machines: DEC VAX, AT&T 3B20, Sun, Hewlett-Packard 9000, AT&T 3B2, and Wicat. Experiments are also underway with IBM 370, Perkin-Elmer 8/32, and Apollo computers. The machines run a variety of UNIX systems, including AT&T System V, 4.1BSD, 4.2BSD, and 8th Edition.

In the process, we found a number of problems with the inter-language communication made necessary by our FORTRAN and C-based system. Specific problem areas included inter-language subroutine calls, character strings, C access to FORTRAN common blocks, distinctions between single and double-precision floating point computations, and dynamic storage allocation.

Introduction

S is a system for data analysis, graphics, and data management that runs under a number of versions of the UNIX operating system on a variety of hardware. S is fully described in a recent book by Becker and Chambers [1].

The primary goal for S is to allow users to perform good data analysis. Judging from the experience of some thousands of users, S satisfies this goal quite well. However, in order for people to be able to use S to analyze data, they must have access to S. Hence it is desirable to have S readily available, on inexpensive but appropriate hardware.

The obvious inexpensive and available hardware today is the "personal computer". Unfortunately, present-day personal machines are not sufficiently powerful to support the computations underlying S, and hence would not succeed in allowing the user to perform good data analysis.

[†] UNIX is a Trademark of AT&T Bell Laboratories.

Interest in wide availability must not compromise the user's ability to analyze data.

Luckily, the general trend in computer hardware is for more power at less cost, and the current selection of professional workstations is the manifestation of this trend. Modern workstations combine computing power, large amounts of addressable memory, and quick and consistent response time, and often come with the UNIX operating system. Many of these workstations also have provision for bitmap graphic displays. These machines not only provide an excellent environment for S, but they also have the potential for providing better understanding of data through dynamic graphic displays.

A Bit of History

The current implementation of S was affected by its historical development. In particular, S was developed with portability (between operating systems as well as between hardware) as a major goal. In the development period for S, and to a large extent today, FORTRAN is the language of choice for portable numerical algorithms. We had experience implementing a portable device-independent graphics system [7], and the statistics research departments at AT&T Bell Laboratories had developed a large library of FORTRAN subroutines for statistical computation.

The first implementation of S, in 1976, was done on a Honeywell mainframe running the GCOS operating system. Even with great attention to portability issues, there were a number of very difficult problems in writing an interactive system in FORTRAN. Most components of S were written as FORTRAN computational algorithms; when a concept was very difficult or impossible to express in FORTRAN, it was encapsulated in a separate module and implemented in a machine-dependent way. Some of these primitive operations were: user interrupts, dynamic storage allocation, and dynamic loading of user-written functions.

In an interactive system, it is important for users to be able to interrupt printouts, computation, or other operations. Dynamic storage allocation is necessary so that there are no hard limits to the size of computations that S can carry out. S asks the operating system for the amount of memory it needs to carry out any particular computation. Dynamic invocation of user-written functions is a primitive because it reflects another goal of S—extensibility. S was designed to allow users to extend its operations and data structures [2]. As new computations or data structures are encountered, the system can be extended by users. (Functions in S are much like commands in the UNIX system; except for a few kernel primitives, everything is a user program. Of course, like the UNIX system, S comes with a wide variety of functions.)

Our experience on GCOS, and a subsequent port of S to the UNIX environment in 1980, convinced us that there were many difficult issues in providing a portable interactive system such as S, even though most of S code was completely portable. Although the concept of operating system dependent primitives allowed us to contain the amount of work necessary to move to other operating systems, but there was still a great deal of system-specific knowledge necessary to write those primitives. In addition, the seemingly trivial effort involved in writing "shell scripts" (or their equivalent in another operating system) became quite large when multiplied by the number of scripts necessary for adequate installation and maintenance of a large system. Organizing the source and object code was difficult without a hierarchical file system. It also became necessary to understand subtleties of the loader in order to make dynamic loading, library creation, and independent linking of functions possible. It was not always easy to get appropriate ASCII control characters to graphics terminals or to avoid buffer overflow on slow graphics devices (such as pen plotters) due to half-duplex communications.

With quite a bit of effort, we solved all of these problems on GCOS and UNIX. However, as time went on, we had a growing awareness that the UNIX system provided a much more hospitable environment for interactive computing than GCOS. It also became apparent that porting the UNIX system was a likely event. We decided to adopt a form of "lazy portability" for S, in which we worked on matching S to the capabilities of the UNIX system while others ported the UNIX system to new hardware. In particular, this decision allowed us to develop a large set of shell scripts, make files, and C-based primitives to fit S into the UNIX environment. As it turns out, we guessed right—whenever someone ports the UNIX system, S can go along for the ride with little effort. Of

course, we had really just moved the portability issue to a different and easier environment: how to be portable among different versions of UNIX and different hardware.

UNIX Experiences with a Multi-Language System

The following approximate statistics should help give an idea of the size of S:

35000 lines of Ratfor	1.3Mb Executable program
2000 lines of C primitives	1Mb online documentation
7000 lines of C utilities and applications	
1000 lines of Shell Scripts	1Mb Object Libraries
2000 lines of makefiles	300Kb data sets
2000 lines of include files	2000 Source files

We have had experience in porting S to the following machines and variants of the UNIX system:

Hardware	Operating System
DEC VAX 11/780, 750	BSD 4.1, 4.1c, 4.2
DEC VAX 11/780	System III, V
DEC VAX 11/780, 750	8th Edition
AT&T 3B20S	System V
HP Series 200 (MC68000)	HP-UX (System III)
SUN 100 (MC68010)	4.2BSD
3B2-300 (WE32000)	System V
Wicat 150 (MC68000)	7th Edition
Perkin-Elmer 32/30	7th Edition
HP Series 500 (HP 32-bit chip)	HP-UX (System III)
Apollo	AEGIS
IBM 370	UNIX/370
DEC VAX 11/780	32V
DEC PDP 11/70, 11/45	7th Edition

When moving S from GCOS to the UNIX system, one of the major decisions we made was the basic choice of programming language. Because of the large amount of available computational code written in FORTRAN, we decided to use that language. However, we decided that the primitive operations of the S system should be implemented in C. C provides the natural linkage with the underlying UNIX operating system calls.

There was no way of providing primitives in FORTRAN, and the concept of translating 35K lines of FORTRAN into C was also unappealing. This left us in a situation that required inter-language subroutine calls between C and FORTRAN. The *f77* compiler document [3] described the linkage between FORTRAN and C, hence in our initial UNIX implementation of S, we were able to write the S primitives quite reasonably. Unfortunately, in our later attempts to move S to other UNIX environments, things were not always so good.

Perhaps the major problem in getting S to run in a new UNIX environment is the prevalence of bugs in the FORTRAN compiler. Since the UNIX system itself is written in C, the C compiler is usually fairly well debugged. However, since there is normally little or no FORTRAN code in a UNIX system, the FORTRAN compiler is likely to be less well debugged.

A more subtle difference is that a FORTRAN-77 compiler is not always the *f77* compiler. In particular, the FORTRAN-C subroutine linkage may not resemble that described in [3].

For example, given a FORTRAN subroutine named "abc", various compilers generate external symbols named "abc_" or "abc". Occasionally a C program in S will need access to a value stored in a FORTRAN common block; common block names for a labelled common named "mycom" include "mycom_", "mycom", and "/mycom". Note that the latter name is one that cannot be accessed by a C program, since C identifiers cannot contain slashes.

Another problem with common blocks is due to the "ambiguous" nature of a FORTRAN common block relative to C externals. FORTRAN programs may have any number of declarations of a common block, but the storage is only allocated once. UNIX System V, for example, only allows one definition of an external, although there may be many references to it (using an explicit *extern* declaration). Using "ld -r" on a FORTRAN program has caused the resulting common block to look like a C definition. Loading together the output of several "ld -r" runs makes the loader think that the common block is being multiply defined.

We decided early on that we did not want to use FORTRAN input/output facilities. Not only were they in a state of change as FORTRAN-66 evolved into FORTRAN-77, but they were not flexible enough to dynamically format data (like the UNIX routine *printf* for example). In retrospect, this was a good decision, since many of the reported problems with UNIX FORTRAN have been in input/output.

Another difficult area is in the treatment of character variables. Although C and *f77* use identical representations for characters, several other FORTRAN compilers use another, incompatible representation involving descriptors (giving string length, and a pointer to the string, for example) rather than simple null-terminated sequences of characters.

An interesting problem is that compilers that strictly enforce the FORTRAN-77 standard [6] make it difficult to hide information. In particular, the standard does not allow multiple declarations of identifiers, even if the declarations are identical. In S, we have "include" files that provide declarations for common blocks. If the user (or a program) includes a declaration for something that was already declared (identically) in an include file, some compilers generate a fatal error.

There are other instances of problems that fall in the cracks between *f77*, *ld*, and C. For example, we have a primitive that interfaces between FORTRAN and the system routine *malloc* to provide dynamic storage allocation. (The interface gives the FORTRAN program a subscript for a vector in a common block which addresses the newly-allocated space.) In several instances there were problems with the alignment of storage caused by mixing FORTRAN and C. *Malloc* assumes that its chain of storage is stored on even-byte boundaries (it uses the low-order bit as a flag to tell whether the block is occupied or not). An odd-length *f77* common block was loaded prior to *malloc* causing the static pointers declared in the C program to be misaligned. No one program was at fault: C knew that it never allocated space of an odd length so it never requested that the loader force alignment; *f77* requested proper alignment for its common blocks, so it didn't bother to fill blocks out to long-word boundaries.

S presently owns a copy of *malloc* which has been modified so that it can free large blocks of storage that may not have been reclaimed for various causes. This can cause problems on machines with different dynamic storage conventions; it may be necessary to use special versions of *malloc* designed for a specific machine.

Another interesting problem arises when trying to write a C function that is callable as a FORTRAN real-valued function. In [5], Kernighan and Ritchie state:

"Because *float* is converted to *double* in expressions, there is no point in saying that *atof* returns *float*; we might as well make use of the extra precision and thus we declare it to return *double*."

Unfortunately, for machines that use the IEEE floating point format, this causes a problem. C functions that are declared *float* still return *double* results, and must be declared double-precision in the FORTRAN calling routine. The problem came up when debugging a C-based random number generator; I have not been able to write a C program that acts as a real-valued FORTRAN function on machines that have different representation for float and double exponents.

More Mundane UNIX differences

A number of more mundane problems arise when attempting to port a large collection of software from one UNIX system to another. In no particular order:

- Some UNIX versions have the capability for setting a maximum size on the files that users can create. That limit may prevent some of the large S libraries or executables from being

created.

- The S installation procedure allows the user to trade off the size of the S executive against the number of separate executables that S may invoke via *forklexec*. Unfortunately, systems with large amounts of main memory but without paging may unnecessarily restrict the maximum process size. This has occurred in a single-user workstation with 2Mb of main memory, yet a maximum process size of 256K. It required reconfiguration of the kernel to get around the restriction. Another problem is that some systems have a limited and hard-to-change swap area, and can have difficulties running very large programs.
- When maintaining a large collection of files, shell metacharacter expansion may cause an argument list that is too long for *exec* to handle. In some cases the limit is quite small, for example 5K characters. This may make it difficult to do certain operations such as creating a large library as in *ar cr lib `cat filelist`*.
- Different UNIX systems have different mechanisms for constructing object file libraries (archives). Seventh Edition systems required the library to be in topological order (typically attained by using *lorder* and *tsort*). Berkeley-based systems have *ranlib*. System V has an automatic archive directory mechanism. It requires a bit of head-standing to write a portable library creation script.
- Commands that have been fixed in one version of the UNIX system may still be "broken" in others. Some systems interpret "*test -w directory*" to mean "test whether a directory can be opened by a user for writing" rather than "test if a file can be written in the directory."
- Seventh edition derivatives may not allow "#" as a shell comment character.
- Some commands still have relatively small wired-in tables left over from the days of small memory. For example, *make* may run out of space with very large makefiles, *f77* has easy-to-exceed limits on the number of symbols, etc. Eventually it would be nice to eliminate fixed-size tables in the commands.
- Some workstations are configured with very small /tmp directories and have trouble dealing with large archives.
- The *echo* command differs from system to system. System V has *echo* built into the shell; the Korn shell, *ksh*, uses an alias for *echo*. Some systems allow C-like escape sequences in *echo*, others do not. See [4] for a description. The point is that such differences make it difficult to write portable shell scripts.

Conclusions

What is the significance of this work to the UNIX community? As UNIX grows and becomes widely available, there are likely to be other large applications written in FORTRAN that will be linked with C for its ability to customize the application to the UNIX environment. The same problems that S has experienced are likely to crop up again and again.

References

- 1 Richard A. Becker and John M. Chambers, *S: An Interactive Environment for Data Analysis and Graphics*, Wadsworth, Belmont, CA, 1984.
- 2 Richard A. Becker and John M. Chambers, "Design of the S System for Data Analysis", *CACM*, May 1984.
- 3 S. I. Feldman and P. J. Weinberger, "A Portable Fortran 77 Compiler", in *UNIX Programmer's Manual*, Seventh Edition, Vol. 2B, Bell Laboratories, January 1979.
- 4 Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
- 5 Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
- 6 American National Standards Institute, *FORTRAN 77*, Standard X3.9-1978.

- 7 Richard A. Becker and John M. Chambers, "On Structure and Portability in Graphics for Data Analysis", *Proceedings of Ninth Interface Symposium on Statistics and Computing*, Cambridge, MA, 1976.

The Dynamics of a Semi-Large Software Project with Specific Reference to a UNIX[†] System Port

Brian Pawlowski

Alan Filipski

Motorola Inc.
2900 South Diablo Way
Tempe, AZ 85282 U.S.A.
(602) 438-3441

ABSTRACT

An operating system port is a complex engineering task characterized by well-defined goals and "easily" measured conformity to specification (the source system) and thus can provide insights into the dynamics of software engineering and project management. In this paper, we illustrate a few practical principles of engineering and management with examples taken from our work, and in the process provide specific guidelines for the timely execution of a UNIX System port.

1. Introduction

In February 1983, the UNIX Project Team at Motorola Microsystems undertook the AT&T-sanctioned port of UNIX System V to the M68000 microprocessor. Fourteen months later our port was accepted by AT&T and, as of this writing, remains the only AT&T validated UNIX System V port. We accomplished the task entirely in-house.

A software project is a dynamic entity. A project advances towards its goals, personnel changes occur through attrition and reorganization, and activities are often drastically modified in response to more immediate market demands. A well-engineered project results in a synergistic relationship between these various activities. A look at these can shed light on some of the interrelationships and motivating forces shaping the project.

In this paper, we present some principles of software engineering with examples taken from our work. The topics we address are:

- Scope
- Scheduling
- Resources
- Staffing
- Techniques and Approaches

2. Scope - Does Once in the Morning Really Do It?

Define the Scope of the project at the outset. Define not only what should be included but what should not be included. It may be advisable to contract the scope if it becomes apparent you have bitten off too much, but expanding the scope mid-project is deadly. Save it for the next project. Babbage never finished his calculating engine because his plans kept becoming more elaborate, until he died of old age in mid-project. Tesla kept losing financial backers because he habitually expanded the scope of his projects and requested more money to continue.

The scope of a project determines the activities to be performed within a project. The definition of scope takes place at the start of the project in its *study phase* and includes: The goal or goals of a project, expected products, problems addressed by the project and problems which lie outside the project.

[†] UNIX is a trademark of AT&T.

2.1 The Goal of a Project A clearly expressed goal simplifies the determination of how to achieve that goal. An exact exposition of a project's goals is the first step performed in the study phase of a project.

An advantage a project such as a UNIX system port has over many other projects is that the final product is functionally well-defined. In most respects UNIX is UNIX, wherever it is hosted. However, even when the product is well-defined, the design and implementation strategies which can be used to produce the product are numerous. The proliferation of differing implementations from previous ports of the same version is a testament to the unbounded imagination of the human mind.

Motorola was commissioned to produce a UNIX System V port to be verified by AT&T. As part of the development agreement we were given a document by Bell Labs called the *Port Acceptance Criteria* (PAC) which described how our port would be judged [5]. This document indicated that the port acceptance process would involve both functional and source level comparisons between the original implementation of System V on the VAX 11/780 and our ported version. In this respect the technical aspects of the project took on a new dimension and resulted in our team performing a *true* port of the UNIX operating system. The PAC document also contained some general guidelines for our work, for example, the reasons for all source-code changes should be made clear with comments. An appendix contained a listing of the names of all source release files (including documentation) with an indication of their degree of portability and whether or not they were required for port acceptance.

In light of our agreement with AT&T and Motorola's intended marketing of the product based on compatibility with UNIX System V OS, the following statement encompassed the goals of the project:

The UNIX System V operating system will be ported from a VAX 11/780 to the M68000 family of microprocessors. Changes to the system will be made to address machine dependencies and performance requirements. Compatibility with the VAX version, both functionally and at the source code level, will be the foremost goal of the project. Where compatibility is impossible, extensions are preferred over base modifications.

2.2 Scope Issues Despite guidelines and clearly stated objectives for the project, as to the scope of the port. As it turned out, the portability classifications given in the PAC were not always useful since they tended to overlook many portability problems, particularly in the utilities area.

For example, `errpt(1M)`, `fuser(1M)`, `sar(1M)`, `dc(1)` and `sh(1)` are listed in the PAC as being source-code portable, but we found items that had to be changed in each before they would run on the target system. `Errpt(1M)` depended upon the format of our error message packets; `fuser(1M)` made MMU dependent assumptions; `sar(1M)` depended upon how information was returned by our block devices. Even the innocuous `dc(1)` was found to be non-portable because of the "A0/D0" bug (described later). The Bourne shell presented portability problems due to a M68000 limitation on the handling of address faults related to user stacks in data space.

These problems, which were identified after the port had begun, were directly related to the problem of portability and needed to be addressed by the project team. However, the scope of the project was not defined rigorously enough to prevent discussion of additional tasks to be performed during the port. This was particularly acute in the area of Non-goals and Non-problems, *i.e.* those areas specifically **not** addressed by the project.

2.2.1 Problems Addressed The following problems areas were anticipated in the port:

- The HIBYTE/LOBYTE problem.
- Compiler dependencies.
- Design of the MMU interface.

These items were easily identifiable as portability problems and were addressed by the project. At the outset of the port there was still a question as to whether all of the existing portability problems had been uncovered, especially in the utilities area which, ideally, should be completely portable. A

systematic search was performed for other non-portable constructs.

Lint(1) was run with the option string "abhuvxp" on all utility source code. This option string suppresses everything except possible portability problems. Even with this suppression, lint(1) generated an enormous number of complaints, mostly relating to pointer casts and other slightly tainted uses of pointers.

In concert with the lint(1) exercise, a grep(1) source code scan on the utilities was also done. We searched for all occurrences of the strings *word*, *byte*, and *union*. This search was done with a known problem in mind, the *hibyte-lobyte* or *NUXI* problem, i.e. the known difference in significance ordering of bytes between the source and target machines. Though we knew it would be a problem, we were unsure of all its possible manifestations. The *hibyte/lobyte* problem is a good example of the interaction of architecture and the porting process. Since most of us had never before worried about such DEC idiosyncrasies, (It took several patient explanations by one of the more avuncular members of our team before we understood why unions, for example, could cause the problem while masks and shifts could not.) this grep(1) search turned out to be useful. Although most of the findings were false alarms, such as the word *password* in the source, a number of real problems were discovered.

Following an initial analysis of the grep(1) data for *hibyte/lobyte* problems it was noticed that many areas which were machine dependent had already been isolated. These machine dependent areas were bracketed by conditional compilation directives with code inclusion based on the target machine. The machines defined included: VAX 11/780, PDP-11, AT&T 3B20S and the IBM 370.

2.2.2 Non-Goals and Non-Problems A useful idea in delimiting project's scope is the concept of non-goals and non-problems, i.e. those areas which are specifically **not** be addressed by the project. Two incidents within the project illustrate some of the snares lurking when non-goals are insufficiently defined.

The first incident occurred during the analysis of the lint(1) data where it was seriously suggested at one point by some members of our group that we massage the code until all lint(1) complaints were gone. This would have been a Herculean task. Following a marathon lint(1) analysis session we realized that the only way to maintain our sanity (and meet our deadline) was to ignore most of the lint(1) complaints - they were obviously ignored by the originators of the code.

In the words of lint(1) itself, many items were tagged "may or may not be a problem", depending upon the compiler used. This Delphic statement could only be resolved when the target compiler was available. The compiler we wanted and eventually got was Bell's SGS II cross compiler for the M68000, but at the time it was snarled in contractual red tape. In order to proceed in the project, we assumed (1) a reasonable target compiler, and (2) the items flagged by lint(1) would not be a problem. These assumptions turned out to be true.

The initial analysis of the lint(1) data was in keeping with the goals of our project. However, the results were so indeterminate as to be meaningless to our immediate work. A curious phenomena occurred in reaction to the indeterminacy of the lint(1) data. Because of the lack of direction from lint(1) some project engineers began discussing problems obviously outside the scope of the project: "I never did like the way they did this in UNIX" and "I can rewrite this code the way it should have been in the first place." To nip this wandering tendency in the bud we chose to do a strict port of the UNIX operating system. The alternative was to get sucked into a quagmire of pointless, interminable fixes. Specifically we observed the following course of action: Port the AT&T utility code, bugs* and all, do as little as possible to get it to run, and report bugs to AT&T and incorporate their fixes as we get them. During the acceptance testing our decision to severely limit our changes to the UNIX system proved wise.

* A Note on the Portability of Bugs: some are and some aren't. Of particular annoyance to us, of course, were the ones which were symptomless on the VAX but malignant on the M68000. For example, in the roff code, there was a structure element, which acted as a sentinel, declared as *char zz*; in one place and as *char *zz*; in another. On the VAX, this was innocuous, since all structure fields are padded to four-byte boundaries on that machine and the *char* takes up as much space as the pointer. The 68000 SGS II compiler, however, pads to two-byte boundaries. The *char* type is allocated two bytes and the pointer is allocated four, with the result that all the data items following the sentinel (a string table used for terminal dependent transformation of text at output) are

Although we did not realize it at the outset, there was plenty of real work to be done in the area of utility porting without us making more work for ourselves. Nearly all makefiles had to be rewritten to operate in a cross environment and certain highly machine-dependent utilities such as `sdb(1)` required extensive work. If we had maintained our original attitude the task would have taken much longer and our acceptance would have been jeopardized.

A second more serious incident impacted the kernel port. Following the resignation of a member of the kernel team we checked the changes he had made to the source code. The `diff(1)` utility was used to provide a line-by-line comparison of the original kernel modules and the ported versions. The results showed that **every** line had changed in a vast number of modules! This was hard to believe as some of these modules were almost entirely portable and the person was **not** on the team long enough to do all that work. We determined that as part of the "porting process" he had run the source through a filter to reformat it the way he preferred it. Since part of our verification by Bell would be the determination of source file differences, it was necessary to reinstate the old versions of the file and extract the real ported code from the reformatted versions. Fixing these files considerably delayed activities in the kernel port.

Following these incidents, a grass-roots principle formed within our group to guide our work. Its most eloquent expression can be found in the form of *Berson's Rule*:

It's been my experience that doing the minimum work necessary to complete a job often results in meeting a deadline. It also leaves time to grab a beer with Pawlowski after work.

This does not mean doing the shoddiest possible job; it just means that, "if it ain't broke, don't fix it." During our port there were cases when, even if it were broken, it was better not to fix it. The "Port Task" is defined as that work *necessary* to bring up System V on a M68000 machine. Thus, in *Berson's Rule*, minimum work necessary is not to be confused with the common lazy man's definition of minimum work. Rather, it is more akin to the efficiency interpretation of work in a system such as a steam engine: The minimum work necessary to effect the desired result (mechanical work), all other internal work being so much hot air.

2.3 Documenting your Scope Two documents are produced at Motorola during the study phase of a project which delimit its scope: *The Functional Specification* and *The Engineering Specification*. The Functional Specification describes the products to be delivered and their functional attributes and covers performance requirements, intended users, packaging and compatibility. Also included is a description of *Non-goals* which we've just discussed. The Functional Specification is occasionally produced by Marketing as a detailed description of a proposed product. The Engineering Specification details the engineering steps required to produce the product and contains design philosophy, system interface descriptions, physical constraints, anticipated problems and tradeoffs made. The Engineering Specification additionally provides schedules, tasks and milestones for the engineering work.

off by two bytes. The characters output from `nroff(1)` were shifted one position in the ASCII map and separated by control characters. The `nroff(1)/troff(1)` system contains about 180 files and working on this problem consumed at least ten man days just before we were scheduled to send System V/68TM to AT&T for approval. This bug very nearly delayed submission of our ported system.

Another amusing bug was the "A0/D0" function return problem. On the VAX, all function return values are left in the R0 register by the compiler. Hence it does not matter whether a function is declared, say, to return an `int` or a pointer, the return value will always be in R0. The 68000 compiler, however, returned `ints` in the D0 data register and pointers in the A0 address register. Mismatched function declaration/invocations thus caused the return of garbage for us. (Actually it was later found that the values returned were typically on the order of '0', '1' or '2', typical integer values left over from previous temporary calculations and function calls). The most common occurrence of the bug occurred as `"ptr = (char *)calloc(sizeof(buf));"`. The explicit cast suppressed any compiler warning and we allocated buffer space consistently in the lower part of the writable text segments! The cause of the bug was only discovered after Read-Only text segments were implemented and we experienced segment violations trying to write to write-protected segments. It is perhaps a testimony to the robustness of UNIX system that programs actually ran fairly well in this state!

At Motorola, these documents are part of *The Product Life Cycle* [1,8]. This document describes the phases of a product and the activities and results of each phase. It is a first approach at nailing down the details of any product. Most large organizations have similar documents which are invaluable in defining the scope of a project.

3. Scheduling

Scheduling is the least understood and least successful activity in a software project. Experience teaches that a project usually will **not** be completed on time. Why is this?

Four factors contribute to missing deadlines in a project:

- Projects which start out small often end big.
- The end result is not what was originally planned.
- The scheduling of a project is performed with little regard for any reasonable task breakdown corresponding to the problem.
- Projects suffer from inadequate tracking and a slip is not recognized until it's too late.

Perhaps it is human nature to want to continually improve, add to and tune things. What can start out as a simple exercise can end up as a huge project. As a project progresses, capabilities are added to the original specification because "it's a neat feature" or "it will only take a couple of hours and make it more flexible and improve performance." It rarely takes just "a couple of hours." A project with inadequate specifications or poorly thought out schedules with few milestones are particularly prone to unplanned growth.

Like the shifting sands in a desert, shifting goals can often obscure the way to one's destination. Although a project at its outset may have a well-defined goal, this goal can change over time, as market demands typically drive a software project. Many projects fail to see the light of day as continually shifting goals end in an abrupt cancellation for failure to produce a marketable product.

The most common heuristic for producing a schedule is to pick a delivery date and work backwards. Although experience can aid a manager in making an initial gut-feel guess on a delivery date, he should reserve the right to revise it. Schedules are most naturally and accurately developed when they are allowed to grow from task breakdowns, task duration and determination of task dependencies.

Once a schedule based on delivery date is pulled out of the hat, it is often not noticed until the project is supposedly near completion that there will be a slip. Schedule slippage is hard to recognize without adequate milestones. Milestones grow naturally from task breakdowns, but of course this planning step is skipped due to the tight schedule imposed by the delivery date. Perhaps proper scheduling is avoided by those presented with a delivery date to meet because they don't want to know beforehand how much they'll slip.

Good scheduling is possible but requires work. The work put into scheduling at the outset is not time wasted but time gained. Whether you put the time at the outset or throughout the course of the project you will perform scheduling functions. If done at the beginning at least you can choose a more effective approach to the solution of an engineering problem. On our project we produced detailed schedules for all aspects of the port. These schedules proved invaluable in manloading the project, determining the impact of changes and additions to the schedule, and justifying realistic needs for additional personnel and resources.

We recommend the following steps to produce a good schedule:

1. Define the goal of the project.
2. Partition the problem.
3. Map tasks to the partitioned problems.
4. Estimate the duration of the tasks.
5. Determine of task dependencies.

3.1 Defining the Goal of a Project Since the goal of an engineering project usually exists before the need for scheduling, it is important to rework the goal as necessary to clearly define the "deliverables" of a project. The Functional Specification becomes an important document at this

point. By clearly defining the goals of a project and its products, we can intelligently address the problems which need to be tackled in the engineering phase. See *The Goal of a Project* under Scope.

3.2 Partitioning the Problem Prince Wen Hui entered the kitchen to find his servant butchering an ox. The cook's cleaver rose and fell with a singing sound and the ox seem to fall apart as if of its own accord. The motions were smooth and sure and the cleaver passed through the ox never once catching in a joint or sinew. When finished, the cook wiped off his cleaver and returned it to his belt. Turning around to see the prince he bowed low. The prince helped him to his feet and said, "How can you so effortlessly butcher an ox never once hitting bone, never once having to remove the cleaver from cartilage or joint?" The cook answered, "A good cook needs a new chopper once a year - he cuts. A poor cook needs a new one every month - he hacks! Their cleavers grow dull by hitting bone and sinew. I have had the same cleaver for nineteen years and have never had to sharpen it. When I get an ox, I size up the beast and reflect that external appearances are not always what they seem. I strive to see the ox for what it is. When I understand this then do I begin to cut him apart. I do not swing my cleaver as a man does an axe but rather feel the natural separations of the whole. So it seems the beast falls apart without effort, as the blade follows the natural divisions of the carcass. I do not cut up the ox but rather the ox allows my blade to pass through. So have I been able to keep this cleaver for nineteen years." [2]

One of the first tasks in a project is the partitioning of the problem. This is often a difficult undertaking as many breakdowns exist for a given problem but not all are good. The effort put into this phase of the project can reduce the headaches down the road. Structured techniques for program design have parallels in project management and provide an aid to the engineer in tackling a project.

For a UNIX system port the gross partitions of the engineering activities are:

- Kernel Port
- Compilers, Libraries and Include Files
- Utilities

3.3 Defining the Individual Tasks A good partitioning is characterized by the ease of mapping the individual tasks to readily separable portions of the project.

There are three factors which lead to this conclusion:

- Separable, independent tasks decrease the need for communication among teams within a project.
- Well-defined tasks which closely map the problems allow for easier integration and testing.
- Project tracking is enhanced by well-defined, independent tasks. Intelligent partitioning allows early detection of schedule slippage and tagging of project "hot spots" requiring immediate attention.

Further divisions of a selected task should reflect the natural boundaries for that task. We have found it helpful to avoid assigning priorities to tasks until all the tasks have been determined.

The level at which you stop partitioning a given task is a judgement call. Since an initial task breakdown is later used to create milestones within a project schedule, a granularity on the order of a week is often sufficient. Dividing tasks in such a manner allows for fine tracking of the project and fits in well with most organization's requirements for weekly status reports. Another guideline to partitioning a task is the analysis of the description for that task. Complex descriptions reflecting multiple activities often indicate that the task breakdown is not sufficient.

3.4 Estimating Task Durations Once the tasks have been delimited each is assigned a duration. Time estimates are based on perceived task complexity and the skills of the assigned engineers. The duration of each task is most naturally expressed in man-days (man-months, man-years?). However as Brooks so neatly puts it: Men and time are not interchangeable [6]. When assigning more than one person to a task, consideration must be given not only to the skill set of the engineers but also to the coordination overhead needed to effect that task. For a medium-sized project, having more than four engineers on a task is possibly an indication that: (1) the tasks have not been partitioned enough or (2) the tasks do not map well to the natural pieces of the problem. When faced with estimating the

duration of tasks nothing beats experience. If the project content is new to the engineering group experience aids in making "best guesses" on task durations.

3.5 Defining the Dependencies A project schedule is straightforward to derive given good partitioning and accurate times for the completion of each task. The only missing information at this point is the dependency structure among the tasks. For a UNIX system port an obvious dependency is that all compilations for the target system depend on the availability of the target compiler. For sanity's sake dependencies should map to the first task depending on a given task. All tasks following the first dependent task are then naturally dependent by transitivity.

4. Resources

In an ideal world, the Functional Specification comes first and then you decide what resources you need to accomplish the task and how long it will take. In this world, however, function is often driven by the competition, schedules imposed by your boss thrice removed, and resources determined by political considerations within your organization. Nevertheless, the task needs to get done.

4.1 Staffing and Manloading Considerations Often, a staff is inherited. Occasionally, it is built up from scratch. In the first case, you may not have the skill set you want. In the second case, it may take months to get a team on board and up to speed. In any case, your staff is your most important resource and building and maintaining a good staff is the key to a successful project. For our project most of the team was in place. Turnover and lateral shifting between our group and others allowed us to fill gaps in our skill set.

An important principle of staffing is to fit the organization to the project as much as possible. With constraints such as experience levels and existing organizational interests, this is often difficult to do. But the project will suffer if it is not done. For our project we had: the project leader, a compiler team of one person, four people on the kernel team and six on the utilities team. We had reasonable-sized groups formed to do the utility and the kernel work, but had only one person to do all the compiler and run-time library work. The organization was structured like this primarily on the basis of skills and interests. When the tasks for the compiler and library "team" were laid out in PERT chart fashion, it constituted the critical path of the project which would take far longer than we could afford (the project would be wrapping up about now). Additional help was given to him immediately and soon several experienced compiler people were freed from another project and put to work on this phase of the port. The compilers and run-time libraries were available when required.

One problem which arose with the above team organization was the perceived lack of utilities work to be done early on in the port. Many of the activities depended on availability of the compiler and a system on which to test "portable" utilities. This resulted in some engineers having more time on their hands than they at first knew what to do with. An alternative organization was suggested by another group performing a very similar port to counter this problem. The initial structure was to be: the project leader, five people on the compiler team, four to five people on the kernel team and one person on the utilities team. The utilities person was to perform most of the work we had done at the outset (categorizing, assigning priorities, and eliminating machine specific utilities). As the kernel and compiler work reached a conclusion the team members were to shift to the utilities team to wrap up the port.

One fact of life in the computer industry is personnel turnover. In our case the problem was exacerbated by the scarcity and high demand for "kernel hackers." We had to compete against Silicon Valley salaries and the excitement (and stock options) of start-up companies. We could not depend upon being able to hire many experienced UNIX system software people, so we had to develop a system wherein we could grow our own. This was done by bringing inexperienced people into the utilities group where they could learn C and UNIX system conventions. They could then be moved up into kernel, run-time library, or compiler work. Whether such a move is a move "up" or not is of course debatable and a matter of personal taste, but in fact, many team members regarded it as such and membership in the kernel group was regarded as having some status attached to it. It was sometimes useful to have such a perceived advancement path, however the desire to advance into an already full kernel team by other members of the group often resulted in discontented engineers. Continual phone calls from headhunters looking for engineers with "kernel expertise" often

exacerbated things.

4.2 Tools A project suffers, as well as the engineers, if the proper tools are not at hand. Development tools not only increase an engineer's productivity and efficiency but also make for a better working environment by reducing the frustrations resulting from inadequate tools.

4.2.1 Software Tools One "side effect" of doing a UNIX operating system port is the ready availability of proven software tools, assuming you have it running on a host system [3].

Software tools which are helpful to a project are: Editors, electronic mail, cross-development tools, text formatters, project management tools and source code control programs.

Electronic mail is useful for posting meeting notices, distributing information to teams on a project and general communication. The electronic mail system allowed project members to get in touch with each other quickly. It was more effective than the telephone since some engineers were rarely at their desks, and it provided a copy of the message with a time stamp. The only drawback to electronic mail is that some people involved with the project are typically not on-line. These people are often in marketing and upper management. Perhaps it's a feature?

The UNIX system mail(1) utility was made far more useful for our purposes through a front-end, written by our project leader, that implemented aliases for groups ("util", for example, meant all members of the utility team) and automatically copied certain classifications of mail to a project log. To control the flow of non work-related mail, we established an alias called "junk" which a person could join to receive jokes, trivia, interesting (?) facts, and restaurant reviews from those of us who fancied themselves wits or writers. This often served as an outlet to relieve the tension of those fourteen-hour work days before a deadline when we were beginning to feel like zombie processes.

Text formatters are useful not only for generating hard copy of those on-line reports for people who collect paper (see Electronic Mail above) but are also useful for producing documentation. Again, a UNIX OS port is fortunate to have the documentation on-line in a format suitable for use by the UNIX system text formatters nroff(1) and troff(1). A possible pitfall here, however, is the heavy load the formatters put on the CPU. Response time at the terminal is noticeably slower when they are running and heavy text formatting should be scheduled at night whenever possible. High quality copy can be produced not only by a phototypesetter (which troff(1) requires) but also by a laser printer with the appropriate software. A laser printer is a cost effective tool for an organization creating its own documentation during a project.

Project management tools are an invaluable aid to a project leader for developing schedules and tracking a project. With so many automated tools of this nature available there is no excuse for poor tracking of a project today. Project management tools not only automate the development of PERT charts and similar representations of project schedules but also permit easy updates and generation of new charts as a project schedule invariably changes (usually for the worse). An added benefit of the use of project management tools is that the nature of the input to these tools forces a manager to define the tasks and interrelationships between them in a manner he might not otherwise.

Keeping revision histories is part of the management of a software project. The Source Code Control System (SCCS) provides sophisticated tools for a complete software evolution control system [7]. Our UNIX system guru created a more restricted and fail-safe environment for our source code by providing a set of front-end programs owned by a "project administrator." This increased the user friendliness and security of the SCCS system.

The generation of target systems was automated via the make(1) utility. The modification of the files required by the make(1) utility for cross system generation on the VAX represented a significant portion of the engineering time on the project.

4.3 Hardware Tools Hardware tools are an important part of the software project. Perhaps the most common problem with hardware support is that the primary development system is under-powered for the expected **maximum** workload. When determining the machine capacity for a project don't figure on "normal load" conditions. Time critical portions of a project (e.g. generating a system for release) consistently produce a heavy load environment. For a multi-user software development system three parameters, besides raw CPU horsepower, are important: Available user memory, available disk space for user files and backup facilities

For swapping systems, such as the UNIX operating system, increasing the primary memory to decrease or eliminate swapping can significantly increase performance under moderate and heavy load conditions. Disk space is often severely underestimated. It is someone's law that a project will grow to meet the available resources. Projects will always manage to eat up disk space! The need for good backup facilities cannot be overemphasized. A daily incremental backup with full backups performed on weekends will save a lot of lost time when the inevitable system crash occurs. Don't fool yourself, your development system will crash and you will be lucky not to physically lose your disk.

If you want your project to progress smoothly get the best maintenance contract possible. Ignoring preventive maintenance or signing a "5 day, 9 to 5" second-rate maintenance agreement will end up costing you a lot of time and money. There are two factors which make skimping on a maintenance agreement unwise: (1) Preventive maintenance, system upgrades and repair will most likely be performed during normal business hours. For a good-sized project this could leave quite a few engineers idle. (2) Second class maintenance agreements often result in second class response to problems and second class repair work. We won't discuss the scenario where a project team performs it's own maintenance on it's systems in a project. Unless you're prepared to allocate time in your schedules explicitly for maintenance, don't be sucked into this potentially disastrous situation.

In operating systems work, bringing up your first system on new hardware is tricky. A compatible medium for file transfer between the host and target system is the best course. Unfortunately, we did not have one; we had to dump file systems created on the host from disk to tape, then transfer the tape via a third system to a target system compatible disk. During the busiest times, we did this at least once a day, often twice. Besides consuming engineering time in baby-sitting the transfers, the system was only as good as its weakest link, and hardware problems anywhere along the line could put us out of operation at a critical time. If you cannot arrange for compatible media and must resort to a Rube Goldberg file transport scheme, at least make sure you have back-up hardware. Work quickly grinds to a halt when you cannot incrementally test your modifications regularly on the target system. When a stable target system was achieved, the uucp(1) communication link provided timely data transfers of limited volume.

It is hard to imagine how we would have accomplished this port in time without a high-capacity UNIX system host, like the VAX 11/780. Although it seems ludicrous to us now, it was originally suggested that we perform most of the port work with a PDP-11/34a! Since Motorola manufactured their own development systems, the presence of the VAX in the design engineering group caused uneasiness among "Motorolans". A solid wood door was first installed in the room housing the VAX, illustrating the principle "out of sight, out of mind." Since that time, the VAX has proved to be a valuable complement to our development systems and workstations and has even earned a window in the door to its room!

5. Techniques and Approaches (T&A)

A number of techniques exist which ease the software engineer's burden. We've already covered many which are useful in the planning stages of the project. Scheduling, man-loading and partitioning are general approaches which make the management of a project easier. From a technical viewpoint the more pertinent areas are those which aid in the day-to-day completion of the project. These areas include: Successive refinement techniques applied to software, structured modular design, unit testing, and test bedding of software.

While these ideas have been around for a long time they have yet to experience complete application in the software industry.

5.1 Successive Refinements Successive refinement of software defines an incremental building technique which allows a programmer to implement and test successive portions of a system. Successive Refinement can occur either in a bottom-up or a top-down fashion. Bottom-up implementation often results in a unit testing approach with a number of integration points designated which culminate in the final system. A top-down strategy allows implementation of higher level functions based on incomplete lower level routines which exist as stubs. Related to these implementation strategies are the structured programming techniques of top-down design and bottom-up coding. In the real world, system implementations typically exhibit both techniques.

The most obvious result of devoted application of successive refinement is well-structured, highly modularized code. Why has this type of design gained wide acceptance in the industry? Successive refinement is a design methodology, a tool for an engineer to use in both the design and implementation of a software system. Successive refinement allows an engineer to logically partition a problem in much the same manner as a project is partitioned during scheduling. Any methodology simplifies a task typically results in more productive engineers.

When porting the kernel a number of features not required for our initial minimal system existed only as stubs (e.g. the Interprocess Communication Routines). The support for user process types was performed incrementally. The only type supported initially was dirty segments (shared code and data) which permitted an initial shake-down of the system. Shared read-only text segments were implemented later.

5.2 Modular Design Highly structured systems are more easily understood. Comprehension is aided if the system consists of modular components which are as independent as possible. Isolation of function and data hiding are characteristic of modular decomposition. Poorly designed systems are often characterized by poorly defined modules with many interconnections between the modules and subsequent dysfunction. Inability to upgrade a system is often indicative of a poorly designed one.

Good programs exist on many different machines throughout their lives. Good programs are also portable programs and the UNIX operating system represents some of the most portable system software today. It is written in a high-level language which encourages modularity and comprehensibility and this has had profound impact on the portability of the system. *It is the authors's contention that an operating system port is essentially a maintenance activity.* It is the upgrade of mature software to allow execution in a new environment.

One modification made to the kernel was the decoupling and modularization of the MMU specific code into a driver-like organization. The number of MMU configurations on DEC systems are limited so the design of the MMU interface is of little lasting importance. However, one of the design goals of our project was to produce a *generic* version of System V/68 which could be easily ported to other M68000-based hardware configurations. The port of the MMU code was felt to be the limiting factor in subsequent ports. The isolation and modularization of the MMU specific code will aid in future port efforts of our system.

5.3 Unit Testing Unit Testing is crucial in operating systems work. Debugging an asynchronous, multi-tasking system is one of the more challenging aspects of systems programming. It is advantageous to isolate those portions of an asynchronous system which can be tested independently in a synchronous environment. During the port a number of opportunities arose to test pieces of the kernel and its support programs in a more restricted environment than that provided by a full blown UNIX system.

An initial, minimally functional, boot loader was created for the target system, a M68000-based EXORmacs® development system. It knew only enough about the file system to find the executable system image "/unix". The first program the boot loader brought up, as "/unix", was a simple program that printed the message "Hello world." to the console. The boot loader was incrementally modified to provide more sophisticated and flexible path search capabilities after the initial version was debugged.

5.4 Test-Bedding Software Systems programming is often complicated by the unavailability of a stable target system in which to test and debug the software. This is an aspect of programming which is not often encountered at the applications level. During a UNIX OS port it is a daily problem. You need a dependable system to test your software modifications. The same situation holds true for original design operating systems and low level programs (firmware). The problems are multiplied ten-fold if a newly designed operating system is being brought up on new hardware.

When faced with an unstable test environment, simulators for the target system and use of a similar system for component testing provide viable alternatives. For a UNIX operating system port, the source computer system affords a stable environment for test-bedding software.

Simulators and test bedding software are also useful in performance analysis. When designing a software system, performance statistics are often required to judge the viability the design.

Performance analysis prior to implementation can save a lot of patching and redesign down the road.

The VAX-11/780 successfully proved itself as a test bed many times throughout the port. When the design of the MMU driver was first proposed the only objection which arose concerned possible performance degradations. Objections to modular design typically center on performance degradation issues and so to determine the performance degradation a test kernel was created for the VAX which implemented the MMU driver. The kernel was exercised and system analysis results showed approximately a 3% (within measurement error) degradation, which was considered acceptable.

6. Conclusion

We've presented many aspects of a software project and have distilled some practical principles of engineering and management from them. It is often hard to relate *principles* to our day-to-day experience and we hope the examples from our work aid in understanding their application. These principles are not new but are founded in the common experience of engineers of all persuasions.

At Motorola, our software engineering department is still evolving. With the growth of a real Evaluation and Design Assurance group at our facility and the incorporation of methodologies through our recently approved *Product Life Cycle* document we are witnessing the growth of a high **quality** engineering organization. Quality and the personal growth of engineers are welcome by-products of a "well-engineered" project environment. Despite the folk tradition that UNIX operating system work is best left to hackers who are locked in a room with a terminal and fed by shoving pizza under the door, our experiences encourage our belief that adherence to common-sense engineering and management principles facilitates *any* project.

The authors wish to thank the people who took the time to review this paper.

7. References

- [1] *The Product Life Cycle* Internal document, Motorola Microsystems, Tempe, Arizona 1984
- [2] Merton, Thomas. *The Way of Chuang Tsu*. New Directions, New York, 1965.
- [3] Jalics, Paul J., Heines, Thomas S. "Transporting a Portable Operating System: UNIX to an IBM Minicomputer", *Comm. of the ACM*, December, 1983. pp. 1066-1072
- [4] Johnson, S.C., and Ritchie, D.M. "Portability of C Programs and the UNIX System", *Bell Syst. Tech. J.* 57, 6, July-August 1978.
- [5] Hergenhan, C.B., Kretsch, D.J., Wehr, L.A. *UNIX System V Port Acceptance Criteria*. Internal document, Bell Laboratories, Murray Hill, New Jersey, 1984
- [6] Brooks, Frederick P. Jr. *The Mythical Man-Month*. Addison-Wesley, Reading, Massachusetts, 1975.
- [7] Cargill, T.A. *A View of Source Text for Diversely Configurable Software*. Technical Report CS-79-28, University of Waterloo, Waterloo, Ontario.
- [8] Council, Ed. "Experiences in the Evolution of an SQA Organization", 12th Annual Rocky Mountain Regional Conference of the ACM, 1984

How to Teach UNIX Q&A

Panel Members

Bubbette McLeod, Chair
Informatics General Corp.
NASA Ames-Research
MS 239-19
Moffet Field, CA 94035

Jay Hosler
User Training Corporation
591 West Hamilton Ave.
Campbell, CA.

Stan Kelly-Bootle
Author of *The Devil's DP Dictionary*
25 Parkwood
Mill Valley, CA 94941

Bob Nystrom
Momentum Computer Systems International
2730 Junction Ave.
San Jose, CA 95134

Jim Joyce
International Technical Seminars
520 Waller St.
San Francisco, CA 94117

Introducing People To UNIX

Bubbette McLeod

Informatics General Corporation
NASA Ames Research Center
MS 239-19
Moffet Field, CA 94035

Since the explosion of UNIX in the computer world, we are faced with the problem of teaching all kinds of people to use UNIX, and some of this teaching is defaulting to programmers who haven't communicated with anything that doesn't have an RS232 interface in years. We will, hopefully, give you some hints on easing potential users into the UNIX community.

There are two types of users to educate, those already computing who need to be brought onto the righteous path of UNIX and those who have yet to compute. When faced with programmers who are experienced but not UNIX-oriented (and who may have been forced to attend your class), you have to cope with issues such as the rationale behind command line syntax or with remarks like "You call this user friendly?" Some find the idea of an operating system that isn't heavily supported to be quite appalling. With rank beginners you have to deal with explaining what an operating system is, the difference between a terminal and a computer, and defining files and programs. Most importantly, you must confront computer fear, and convince the users-to-be that the computer is a useful tool, not just an electronic ogre who bills them mistakenly for 12 years of *The Poultry Breeders Review*.

To start with, it is important to tailor your classes to your audience. Beginners should not be overwhelmed with too much information at once, but the level of repetition necessary with them is frequently boring to more experienced users. Secretaries need to learn things like *spell*, *nroff* macros, possibly how to use the *style* and *diction* programs. While programmers could probably use these things, they tend to be more interested in programming tools like *lint*, *adb*, *awk*, *sed*, and *vi*'s features for editing programs. When teaching programmers familiar with other operating systems it is valuable to make them aware of the ease with which they can integrate UNIX utilities into their programs and eliminate writing unnecessary code.

An ideal learning situation is a one-on-one student teacher combination, but this is not usually practical. When there are too many people to stand around a terminal and watch, you are often stuck demonstrating how to log in or use a screen oriented text editor on the black board. If the equipment is available hooking a terminal up to several video monitors is a good way to demonstrate logging in and show off the flashier features of *vi*.

Good handouts are very important. People love to have something they can scribble in the margins and take away for future reference. Cheat sheets are a big hit, too, especially if they are tailored to the user's needs. In some cases handouts can teach the class for you. For *nroff*, giving out examples of *unnroffed* and formatted text can be extremely helpful. The users then have templates to consult when doing their own formatting. The wisdom of using *nroff* macro packages can be displayed by showing the same text in pre-*nroffed* form using both raw *nroff* and *-me* formatting commands. The *script* program can be useful for creating handouts by making a transcript of what's going on the screen. You can do things like step the user through logging in, listing directory contents, changing directories, doing *pwd*, etc.

When approaching the neophyte computer user you have to assume they know absolutely nothing about computer. Start at the very beginning. If possible, remember back when you knew nothing about computers. Don't pepper your language with confusing terminology like I/O, parse, and parameter. Be careful not to belittle or ridicule them, no matter how silly their questions or mistakes. If they are too afraid of doing something wrong they won't do anything. If you get the types who say "I'm just too stupid to learn about these computers . . . or I'm so bad with machines . . . or math . . .," be generous with praise, they need all the encouragement they can get.

With the brand new user, many times the first thing you need to do is confront computer fear. It is necessary to first demystify the computer. This can be aided by computer games, communication with other people via the computer (*talk, write, mail, usenet*), and where appropriate a tour of the computer or computer center. Take them around, show them the CPU, the tape drives, explain how a disk works. Somehow seeing the computer in its various components with their own eyes makes the new users much more comfortable, even if they can't recall the next day just what a disk drive is. They are also less likely to call you up and say "the computer's broken" when they've just gotten stuck in *vi* or hit the no scroll key accidentally.

Computer illiterates sometimes tend to be afraid they can break the computer. They honestly, often, believe that if they type the wrong thing the machine will crash, or worse yet, blow up. If their fears on this count are reassured, they will be able to proceed with much more abandon, though there will soon come a day when they wished they could break it!

New users are much more confident working in pairs. That way they are not alone facing the big bad machine, and have someone to compare notes and conspire with.

Using mnemonics can greatly simplify and speed up the learning process. One I am particularly fond of is explaining the redirect signs as being arrows pointing to and from files. It makes the concept of input and output a lot clearer. Control S and Control Q can be explained as control Stop and Control Quick(ly continue); and logging out with control D is simply Control Done.

In teaching text editors it is very important to make the point that blank spaces, tabs, and carriage returns are characters just like letters and numbers, as far as the computer is concerned. Once these concepts are gotten across, users will find it much easier to use editing commands.

In some ways teaching a programmer, be it an RSXer up for conversion, or a UNIX programmer who just wants to learn a few new wrinkles, is a relief. You no longer have to be so careful with your language or go through such detailed steps in explaining things. But, you really have to know your material. If you make a mistake, they'll nail you every time.

With more sophisticated users, you can upgrade the contents of your discussions by explaining how things work, in addition to presenting cookbook instructions.

When teaching *nroff* to sophisticated users, you have two options. You teach them that it's black magic, do it this way and no other or it's bound to fail, and it may not work right anyway. Or, you can teach them that *nroff* really is a programming language, although a primitive one, with very poor documentation, and like any programming language you have to know (or in this case figure out) the rules and follow them.

With people very familiar with other operating systems, comparing the commands from their previous system with UNIX can be useful. Documents like *A Guide to UNIX for VMS Users* (by Michael Urban of TRW) can be a great help.

Occasionally, you will get users who are experienced with other operating systems who are only attending your class because their boss told them to "go learn something about that UNIX operating system." These people will sometimes be resentful of being forced to learn some newfangled operating system. The instructor should be prepared to fend off nit-pickers (language lawyers are one example), know-it-alls, and even outright hecklers. Don't get upset and keep your sense of humor and you will make it through. Some of the questions about why things are done a particular way can be mollified by simply saying "It's a feature."

INTERACTIVITY IN PACKAGED UNIX TRAINING: A MODEST PROPOSAL

by Jay Hosler

V.P., Product Development
User Training Corporation

ABSTRACT

Because using UNIX is a critical part of learning it, packaged UNIX courses should include direct interaction with UNIX. Computer-managed courses commonly fail to provide enough direct experience on the target system. A strategy is proposed for guiding trainee interaction, providing the necessary direct experience without resorting to sink-or-swim pedagogy.

Ten years ago UNIX was unknown outside of Bell Labs and a few universities. By 1980 the software industry had succumbed: companies everywhere were using UNIX for development. Programmers in large numbers had to learn to speak grep. They learned the same way programmers have always learned new systems: by slogging through the documentation, and by watching each other.

The UNIX documentation of 1980 was less complete than its System V counterpart. But its inimitable style was already evident. Then as now, it demanded much of its readers. More than any other aspect of UNIX, the Bell documentation reveals that UNIX was created by and for system programmers.

Faced with learning a system whose documentation was terse and full of inside jokes and jargon, people helped each other. UNIX expertise spread through the programmer community with the help of the buddy system, a minor form of apprenticeship.

The combination of the documentation and the buddy system satisfied the UNIX training needs of 1980.

Then came the UNIX explosion. By 1982, the appearance of early warning signs of UNIX outside the technical community made it clear that UNIX would affect the lives of many more people. But the growth of the user community was overshadowed by a dramatic shift in its makeup: for many, UNIX would provide the first introduction to computers. For the computer novice, neither the documentation nor the buddy system is an adequate UNIX learning aid.

Now, suddenly, we have a cornucopia of products designed to help people learn UNIX. There are books by the dozen about using UNIX and programming in C. There are seminars for everyone in the UNIX community -- users, managers, buyers, sellers, programmers, engineers, and aspiring product developers. But because books and seminars can't handle the load even now, let alone next year, there is also a spate of packaged training products for UNIX. Courses are available on

video cassette, on audio tape, on videodisc, on floppy disk.

It's this packaged courseware that I want to talk about. The topic interests me because I've been making packaged UNIX training for three years. At User Training Corporation, we make UNIX courses using audiodigital(TM) recordings. Our courses replay an instructional session, recorded live, on the trainee's terminal.

But the nature of packaged UNIX training should concern all of us. The economics of the UNIX explosion dictate that most new users who get formal UNIX training will get it from a packaged course, not from a personal tutor or even a flesh-and-blood classroom instructor. Packaged courseware will have great leverage in shaping the skills and attitudes of a boom generation of new users. UNIX is widely reputed among the innocent to be arcane, overcomplex, and difficult to master; the Bell documentation does nothing to dispel this view, which reflects the heritage of UNIX. To some extent, the success of UNIX in the broad applications market will depend on how well the available training overcomes this stigma.

UNIX doesn't completely deserve its reputation for user unfriendliness. UNIX isn't genuinely hard to learn; rather, it's hard to teach -- or at least to teach well. Good training works; a well-trained user sees UNIX as friendly.

Unfortunately, misconceptions are widespread about what makes a training course work. The most damaging of these misunderstandings concerns the training role of interactivity, which refers to reinforcement by practice: a trainee must use UNIX to learn it. In computer training, doing is essential to understanding.

To place this issue in proper perspective: interaction between the trainee and the target system is only one of several attributes that determine courseware effectiveness. If a packaged training course is to work, its design should address each of these additional issues:

TEACHING BY DEMONSTRATION: most people learn a new skill more easily by observing a demonstration than by reading about it or hearing about it. Demonstrations with audio are best: people make better teachers than computers.

COMPELLING PRESENTATION: no training occurs when the trainee is sleeping. If the course fails to hold the trainee's attention, nothing else about it matters.

APPLICABILITY: the training environment should be as much like the target work environment as possible, to minimize the transition from training experience to working experience.

FLEXIBLE LEARNING PACE AND STRATEGY: people march to various drummers. Learning rate varies substantially among trainees. And while one trainee may learn most efficiently from short study sessions combined with short practice sessions, another may need longer study and practice sessions or may take an experimental approach.

GOOD PEDAGOGY: the difficulty of learning a skill depends partly on how many independent concepts the trainee must learn. Good course organization and well-designed mnemonic reference materials make learning easier by providing a framework that can help the trainee remember what he needs when he needs it.

INTERACTIVITY, however, stands out from these other desirable course qualities. Perhaps the learning process can survive poor organization, or poor presentation. But beware the course that isn't connected to the target system! If a course gives the trainee inadequate experience using the system he needs to learn, it fails him in a fundamental way: it leaves him, at the end of the course, ready to undertake his first real interaction -- precisely the hurdle he presumably expected the course to help him cross.

Unfortunately, the issue of interactivity has become the center of a storm of confusion. Many packaged training products, including some that address UNIX, require the trainee to interact with the training system -- but not at all with the target system. For example, the popular computer-based "teaching diskette" courses for end-user applications are "interactive" in that they conduct a dialogue with the trainee. However, their appearance of teacherly concern can conceal a basic flaw: they rarely give the trainee experience using the program he needs to learn.

The purpose of interaction during training is to give instant feedback to the trainee's responses. Computer-based training techniques provide this feedback by exercising programmed control over the trainee's practice sessions. The control strategy can be simple, as when the trainee answers multiple choice questions, or elaborate, as when he interacts with an emulation of the target system.

Theoretically, a computer-based course that judges each response intelligently could provide excellent training. In practice, however, the design task of anticipating what the trainee may do wrong is overwhelming. Consequently, most computer-managed training courses have only a limited repertoire of responses and provide a learning environment completely unlike the target system. The result is often pedagogical hash, as when a course demands that the trainee press the A, B, or C key (instead of the UP ARROW key) to indicate which key moves the cursor up on the screen.

Even courses that emulate the target system are normally very limited, and often fail pedagogically. For example, LEARN, the now-defunct UNIX self-teacher, confuses the trainee by secretly varying his environment. By obscuring the issue of who is in control, LEARN reduces the value of the trainee's practice sessions.

The computer-based courses available in the retail software training market display many weaknesses related to the issue of interactivity. Some effectively prevent the trainee from making errors (recovering from errors is one of the major skills he must learn). Other products adopt a "big brother" approach to interactive testing by assuming that the trainee is using the product under duress -- a self-fulfilling prophecy. High-quality computer-based courseware is surely not

impossible. It seems, however, that the enormous design cost of good courseware has prevented computer-based training from realizing its potential in today's computer training marketplace.

These observations discourage me about the applicability of simple computer-based training techniques to UNIX. Teaching someone to commit acts of UNIX is different from teaching him to tighten bolts on an assembly line or even to use a word processor. A successful UNIX user needs more than a repertoire of commands at his disposal; to know how and when to best use the commands, he must understand the UNIX concepts that underlie them. For example, a user who understands the structure of the file system can readily learn the file-handling commands. But if his conceptual base is inaccurate, he will make errors no matter how well he knows the syntax of the copy command.

In other words, it doesn't make sense to learn or teach UNIX by rote. With its many quirks and distractions, UNIX is a complex new world for the neophyte trainee. Fluency in UNIX demands UNIX experience above all else.

I propose, for teaching UNIX, an approach that produces packaged courses that work. I believe the quickest way to UNIX skill is guided practice on UNIX itself. Interaction during training is to provide instant feedback. A trainee who interacts directly with UNIX receives the best possible feedback, from the system itself, just as he will when he completes the course and goes to work.

I do not suggest that adopting guided practice in place of programmed-learning scenarios will reduce the design effort required to produce a good course. The major task for the designer of any interactive practice session, in either case, is devising a scenario simple enough to limit the trainee's opportunities for error and confusion, then anticipating what can go wrong.

But to learn UNIX, I believe that the trainee should interact with UNIX itself, not a pale imitation. A well-written exercise that anticipates the trainee's troubles, and explains them with humor and patience, is far superior to a "dumb" question-and-answer program. Experience gained answering questions about UNIX, or interacting with a mockup of UNIX, is like the experience a pilot trainee gains by flying a simulator. It can impart the basic ideas. But for UNIX or airplanes, the learning that sticks comes from hands-on experience with the genuine article.

Workshop on TCP/IP and Networking

Mike Muuss and others

Abstract

This workshop will have three distinct phases: First, there will be a mini-tutorial on the TCP/IP protocols (MILSTD-1777 and 1778) and how they work, second will be a case study of several existing IP-based local networks, and finally the panel will discuss various options for building a network at your facility, and answer questions from the audience.

Discussion will be restricted to ISO Levels 1-4 (i.e., TCP, IP, drivers, and hardware) of the protocols, making this workshop most appropriate for systems managers, kernel hackers, hardware people, and communications specialists. No licenses are required for attendance.

Who should attend

People who will need to connect a host to DDN (MILNET) or ARPANET. People who are interested in using the 4.2 BSD network support to build a network. People who are interested in building a mixed-host network and wish to avoid vendor-specific protocols.

Software Tools Users Group Technical Session
Thursday, June 14, 1984
Salt Lake City, Utah

CONTENTS

Avionics Simulation Package: A Large Systems Application in Ratfor
Dave Martin, Hughes Aircraft Company
An Update on the Software Tools Standards Effort
Bill Meine, Sun Microsystems
Ada? Yet Another VOS?
Neil Groundwater, Analytic Disciplines Inc.
A LEX Tool for the VOS
Vern Paxson, Real Time Systems Group, LBL
A Portable TOPS-20-like Implementation Command Parser
Nelson Beebe, University of Utah, Dept. of Physics
Mine Planning Applications in Ratfor
Michael Norred, MINESoft
A Ratfor Implementation of KERMIT
Allen Cole, University of Utah, Computer Center
Future Directions for STUG (open discussion)
Dave Martin, President, Software Tools Users Group, Inc.

Avionics Simulation Package: A Large Systems Application in Ratfor

Dave Martin
Hughes Aircraft Company
P.O. Box 92426 Bldg. R1/C320
Los Angeles, CA 90009

In the past, it was difficult to write and maintain software systems which required the static sharing of data between routines because each routine was required to contain its own description of that data. The presence of an "include" directive has gone a long way toward solving this problem for routines written in the same language. The multi-language case still presents problems, due to the different ways languages specify static shared data (i.e. common blocks vs. structures).

The Avionics Simulation Package (ASP) solves this problem by introducing a neutral data definition format and providing utilities to automatically generate language-specific declarations suitable for inclusion by RatFor, C and FORTRAN routines. A similar mechanism is provided for the specification of symbolic constants in a language-independent fashion. This talk will discuss the use of the Software Tools VOS in the construction, maintenance and configuration management of large avionic simulations.

An Update on the Software Tools Standards Effort

Bill Meine
Sun Microsystems
1333 West 20th Ave, Suite 200
Denver, CO 80234

A standard VOS specification is essential to achieve true software portability. The Software Tools Users Group Standards Committee has elected to first standardize on the primitive subroutines and Ratfor in order to facilitate the development of conformant VOS implementations. The first round of balloting has been completed, and the results were reported at the Washington DC conference. The speaker, Chairman of the Committee, will discuss the status of the second ballot and the plans for future ballots.

Ada? Yet Another VOS?

Neil Groundwater
Analytic Disciplines Inc.
2070 Chambers Road
Vienna, VA 22180

The Ada programming language has been mandated for Department of Defense applications programs. In an effort to provide software engineering tools a support environment is evolving which contains tools similar to those provided by the Software Tools Users Group and Unix [™]. The evolution and status of these environments will be discussed and contrasted.

A LEX Tool for the VOS

Vern Paxson
Real Time Systems Group
Lawrence Berkeley Laboratory
1 Cyclotron Road 46A
Berkeley, CA 94720

The Real Time Systems Group has recently written a Tool for the VOS which is similar to the Lex program available with Unix. Lex takes as input a list of regular expressions, each of which is associated with some action written as Ratfor code, and as output produces a pattern-matching Tool. When invoked, the generated Tool will read in text, and whenever one of the specified regular expressions is matched by the text, the action associated with the pattern will be executed.

The emphasis behind our Lex implementation was that Lex should generate both efficient and fast Tools. We used finite automaton techniques to make the pattern-matchers run quickly and without backtracking, and spent a considerable amount of time developing a compact representation of the automaton so that both time and memory are used efficiently. The result is a powerful pattern-matcher generator which outperforms the Unix Lex.

A Portable TOPS-20-like Command Parser

Nelson Beebe
Department of Physics
University of Utah
Salt Lake City, UT 84112

This talk will cover a portable command scanning package modeled after the TOPS-20 monitor call COMND. The package provides support for writing interactive command scanners with command completion, help, and keyword prompting. Commands may have many tokens and a wide variety of argument types such as strings, real numbers, and file names.

Mine Planning Applications in Ratfor

Michael Norred
MINESoft
13271 West 20th Ave.
Golden, CO 80401

In the course of developing a major mine planning package, the decision was made to use Ratfor. Immediately, a love/hate relationship ensued. The primitive set and library were extended to support engineering applications. Problems in portability arose due to the lack of standards for software tools installations. Overcoming all odds, the package was completed and ported to a variety of hostile operating systems with no change to application code.

A Ratfor Implementation of KERMIT

Allen Cole
University of Utah
Computer Center
3116 MEB
Salt Lake City, UT 84112

KERMIT is a protocol designed for reliable file transfer between computer systems via serial communication lines. KERMIT was originally developed by Columbia University to provide communication between a DEC-System 20, IBM 370-series mainframes, and various microcomputers. Implementations are now available for a wide variety of computer systems. As a result, STUG has selected KERMIT as its baseline protocol for the electronic distribution of its software, newsletters, and other information. The actual protocol specification was discussed at the Washington DC conference. This talk will describe a working VOS implementation.

Future Directions for STUG (open discussion)

Dave Martin
President
Software Tools Users Group, Inc.

The Software Tools Users Group has four major goals:

- Specification of a standard, portable VOS environment
- Production and/or collection of implementations of the VOS
- Distribution of the VOS and its supporting documentation
- Providing a communications path between STUG members

This session will discuss STUG's progress toward meeting these goals and will propose ways in which we may meet them better in the future. Suggestions from the membership will be accepted and discussed.

A Ratfor Implementation of KERMIT

Allen M. Cole

University of Utah Computer Center

ABSTRACT

Kermit is a program which allows files to be transferred over tty communication lines. Kermit in addition can perform a variety of tasks which aid the file transfer process such as terminal emulation and list the names of files on the remote computer. Kermit has been implemented in a variety of languages on a variety of computers. The Software Tools system includes primitives which allow a programmer to write portable programs. Using these primitives, it is possible to implement most of the features of the Kermit program.

1. Introduction

The low cost of personal computers means that many jobs formerly done on main frame computers can now be done on a PC. These jobs include text editing, document processing, and program development. Transferring work from the main frame can only be accomplished if a reliable method of communications can be found. One historically common method of communications, transfer via tape, is generally not available. High speed communications, such as Ethernet, are available for specific machines, specific PCs and specific operating systems.

The solution which requires no special hardware is to use commonly available terminal communications ("tty" communications). Simple forms of tty communication may require no additional hardware or software. File transfer can be accomplished with the following procedure. First, a communications port on the PC is connected to a terminal port on the main frame. Then, the main frame is instructed to "list" a file on the terminal line. Finally, the PC is instructed to copy the characters from the communications port to a file on the PC's disk.

Unfortunately, the output from such a procedure is seldom an exact copy of the file on the main frame. The additional step of extracting the real file is necessary. This is the essence of tty file transfer: simple and inexpensive; error prone and requiring special attention. The reason tty transfer is popular is that often the only alternative is to retype the file by hand on the other machine.

Uploading, the process of sending a file from the PC to the main frame, has other problems. Many main frames do not allow the terminal user to "type-ahead". After one line is entered on the terminal, the user must wait for the main frame to signal that it will accept the next line. Even on systems which allow type-ahead, the type-ahead buffer is often too small for the quick PC. Typing too fast for the main frames may result in a file which is substantially complete, but missing a few lines. One software solution to this problem, known as prompted upload, has a program on the PC wait for the go-ahead prompt from the main frame.

Perhaps the greatest problem with tty file transfer, however, is the problem of transmission noise. Transmission noise is a real problem when using phone lines for file transfer. Even a single character transmission error can break a piece of software. Tty file transfer can be made reliable by using a communications protocol managed by coordinated software on the PC and on the main frame. Such a protocol, the Kermit protocol, allows for reliable file transfer over tty lines. The Kermit protocol was developed at Columbia University by Bill Catchings, Frank da Cruz and Vace Kundakci. This protocol is implemented with two programs. One Kermit program runs on the main frame and another on the PC. Two coordinated programs can solve all of the problems discussed so

far, plus a whole variety of problems which have not been broached.

The disadvantage of Kermit's approach is that it requires twice as many programs. In addition, Kermit has not been written in many portable languages. Kermit is not a gigantic program, however, so the problem is not insurmountable. A simple version of Kermit written in C is a 1200 line program. The Ratfor version of Kermit is approximately 1500 lines.

2. Problems with Tty File Transfer

The various Kermit programs implement the Kermit protocol. The protocol solves the problems described so far. It also solves several other problems inherent in tty file transfer. The following description of the Kermit protocol will be prompted by mention of a specific problem of tty file transfer, followed by the feature of the Kermit protocol which addresses it.

Problem #1:

It is difficult to transfer files with lines longer than the terminal will print.

Solution:

This problem, along with several others, is solved by forming packets from the characters in the file. Characters from the file are accumulated, ignoring line boundaries in the file, to fill a packet. Thus a long line in a file may be spread across several packets. Several short lines may be needed to fill one packet. The Kermit program at the other end of the transmission is responsible for re-assembling a file with the line length of the file on the originating machine.

Problem #2:

Different machines have different methods for representing the end of a line in a file.

Solution:

Most systems print the character pair "Carriage-Return Line-Feed" at the end of each line in the file, or when folding long lines. Other systems, Unix for example, represent the end of line with a single "Line-Feed" character. This translation is not complicated, but is a bother. The solution in the Kermit protocol is to represent the end of a line in an agreed upon way. The Kermit on the local system is responsible for translating the end of line sent by the originating Kermit to the local convention. Placing the burden for adapting the file to local conventions is part of the general philosophy of Kermit. File names, for example, from the originating Kermit program must be translated to something acceptable to the receiving Kermit.

Problem #3:

It may be difficult to have a file printed without extraneous material.

Solution:

On some systems, it is difficult, or impossible, to print a file on the terminal without such extraneous material as line numbers, the name of the file, or a message about the end of the file. On many systems, messages from other users or systems messages may be (perhaps inadvertently) interspersed with the file to be transferred. Kermit packets solve this problem with a beginning of packet character, end of packet character, and packet count. Any characters which are printed outside the boundary of the packet are ignored by Kermit. Any characters mistakenly printed within the boundary of the packet are detected.

Problem #4:

Phone lines are "noisy" and garble characters.

Solution:

Each Kermit packet contains a packet number and checksum. If the checksum is verified, and acknowledging packet is sent. If the checksum indicated corruption, a specified packet will be retransmitted.

Problem #5:

Some systems provide "special" handling for control characters.

Solution:

Some computer systems do such things as expanding tabs, requiring that control characters typed on the terminal be quoted with some other control character, or refusing to pass certain control characters such as "Form-Feeds". Kermit solves this problem by quoting all control characters with a printable quote character.

Problem #6:

Some systems will only allow a line to be entered after they issue a "ready" prompt.

Solution:

The problem of typing too fast for a main frame which expects a human typist is solved by acknowledging the receipt of every packet. If limited type-ahead is available, the originating Kermit can be set such that only one packet, of a specified size, will ever be sent the the terminal line before an acknowledgment is received. On machines where no type-ahead is available, the Kermit end-of-packet character has been set to be the remote computers "ready" character.

Problem #7:

It is difficult begin the process of file transfer. You need to tell two machines different things at the same time. One machine needs to be instructed to "list" the file, the other to "catch" the file.

Solution:

There are two solutions to this problems in different Kermit programs. The first versions of Kermit usually solved this problem by requiring that all reading be done with a time-out mechanism. This prevents deadlocks by allowing the originating machine to retransmit the packet if an acknowledgment is not received within a specified amount of time. This solves the problems, but plays havoc with portability. The second generation Kermit programs have a feature which largely bypasses the problem of synchronizing the two machines. This features, known as "server mode", designates one of the computers (typically the main frame) as the passive "server". All transactions are initiated by the other computer (typically the PC).

3. Implementing Kermit

The Kermit program was originally developed and placed in the public domain by Columbia University. The following is the list of Kermit implementations as distributed by Columbia for November, 1983:

Machine	OS	Language
-----	-----	-----
DEC-10	TOPS-10	MACRO-10/Bliss
DEC-20	TOPS-20	MACRO-20
DEC VAX	VMS	MACRO-32/Bliss
DEC PDP-11	RSX/11	MACRO-11
DEC PDP-11	RT-11	OMSI Pascal
DEC PDP-11	RSTS/E	MACRO-11
DEC Pro-3xx	P/OS	MACRO-11/Bliss
DEC Rainbow	CP/M-86	Asm86
IBM 370	VM/CMS	IBM assembler
IBM 370	MVS/TSO	?
IBM 370	MUSIC	?
IBM 370	MTS	?
IBM 370	GUTS	?
CDC Cyber	NOS	Fortran
Honeywell	MULTICS	PL/I
UNIVAC 1100	EXEC	Assembler
DG MV/8000	AOS	?
PRIME	PRIMOS	PL/I
HP-1000	?	Pascal?
HP-1000	?	Fortran?
Xerox 1100	(InterLisp-D)	
LispM	(Common Lisp)	
Symbolics 3600	Lisp Machine	
Various	MUMPS	MUMPS
Various	UNIX	C
VAX	4.1bsd	
SUN	4.1Cbsd	
IBM 370	Amdahl UTS	
Others	V6, V7, Onyx, etc.	

Machine	OS	Language
-----	-----	-----
Various	UCSD P	Pascal
Terak		
HP-98xx		
IBM PC		
Corvus Concept		
Sage II & IV		
IBM PC	PC DOS	MASM
Zenith Z100	MS DOS	MASM
Compupro	MP/M-86	C?
Z80/8080	CP/M-80	8080 assembler
DEC VT180		
DEC Rainbow (Z80 side)		
DECmate II (CP/M only)		
Heath/Zenith-89		
Heath/Zenith-100 (Z80 side)		
Apple II (Z80 soft card)		
TRS-80 II (CP/M only)		
Osborne I		
Intertec SuperBrain		
Vector Graphics		
Telcon Zorba		
Ohio Scientific		
"Generic" CP/M 2.2		
"Generic" CP/M 3.0		
Atari	DOS	?
Apple II	DOS	Cross
Commodore 64		Cross?

Most of the Kermit implementations to date have been done in low-level languages. The few implementations which have been done in higher-level languages have made such heavy use of special features of a particular language implementation or operating system that no portable version of Kermit exists.

The Software Tools system is a set of primitives which provide such capabilities as file I/O, simple task control, and conversion utilities. We, (Allen Cole and Kendall Tidwell), hoped that it would be possible to write a version of Kermit based on the existing Software Tools primitives. In addition, the similarity of C and Ratfor prompted the belief that the C version of Kermit could be converted to Ratfor with a minimal effort. While bringing up a version of Kermit in Ratfor, we found that at least one more primitive is needed. We also discovered that "hand" translation of C to Ratfor required special attention. The following discussion is in two parts: a description of primitives required to implement Kermit, and a description of the conversion of the C

version of Kermit to Ratfor.

3.1 Primitive Routines

The description of the primitives necessary to write Kermit will be done by listing the Unix primitives required. After the name of each primitive, I will describe the feature of Kermit which requires its use. Some of these features can be done without, others cannot.

fork

The fork primitive is used for the terminal emulation feature. Terminal emulation consists of a fork followed by two tight loops: one copies characters from the port to the terminal screen; the other copies characters from the keyboard to the port. This routine could be substituted with a special purpose reading routine. This routine would not be generally available in Fortran.

stty

The stty primitive is used to set special terminal modes. In particular, type-ahead, and "raw" mode are desirable. Raw mode is the minimum amount of translation available when printing or reading characters. Echoing, special handling of control characters, and printing such things as "Carriage Return-Line Feed" at the end of lines all slow down the transfer process. Kermit is insensitive to most terminal modes, so the only possibly gain from this primitive is speed of transfer.

exec

Exec is used to execute commands used in the Kermit server mode. This routine isn't used in the first C version of Kermit, but is necessary for the server version. If exec is not available, the code for many of the server functions, such as directory lookup and file printing, could be incorporated within the Kermit program.

bit fields

Bit fields are used to mask part of each character as part of the check-summing code. This is essential for file transfer.

alarm

Alarm substitutes for error checking in many parts of the C version of Kermit. If anything goes wrong, the alarm goes off, and a failure is reported. An especially important

function of alarm is to catch a deadlock. Deadlocks can also be detected by a "read-with-timeout" routine.

sleep

The sleep routine is used to wait for a specified amount of time. This is used in Kermit to allow the PC which is receiving a file time to prepare. This routine is not contained in the C version of Kermit, but is critical if Kermit is implemented without a "read-with-timeout" routine.

There are other miscellaneous capabilities required. The terminal emulation feature requires the ability to read from and write to a communications port. This is typically handled by specifying the name of the port to the routine which opens file. Some systems have this feature, some do not.

The list of applicable Software Tools primitives is unfortunately small--just one routine.

spawn

The spawn primitive is similar to "exec". It can be used to execute programs.

Since there are so many required routines are not available in Ratfor, there were two possible courses of action: drop features, or write new primitives. A combination approach was chosen. The following routines were developed for the Software Tools Kermit program. Perhaps some of them should be incorporated into the regular primitive routines. Fortunately, all of the routines are relatively simple.

getbit

Getbit extracts a bit field from an integer. This is often available in the Fortran language, but could be implemented with shifts or arithmetic.

setbit

Setbit replaces the bits in one integer with bits from another. This is often available in the Fortran language, but could be implemented with shifts or arithmetic.

sleep

Sleep waits for a period of time. This function is typically available as an operating system call, but could be implemented with a routine which performs some mundane task for the appropriate number of iterations.

setraw

Puts the terminal in "raw" mode. This is very system dependent. It is fortunate that Kermit will work without this call.

unsetr

Reverses the effects of "setraw".

The other routine necessary for a full implementation is a routine which does "read-with-timeout".

3.2 Translating C to Ratfor

The process of translating a C program to the Ratfor language has many traps for the unwary. While most of the problems might be solved by a translating program, such a program has not been written. The basics of the translating are easy:

1. Put all C global variables in Ratfor common blocks.
2. Adjust the form of the "define" and "include" statements.
3. Adjust the form of the lines by adding "CALL" statements where appropriate and removing the C line terminator ";" where appropriate.
4. Make index variables for each character array.

One subtle source of errors is the difference between C's convention of passing scalar variables by value, and Ratfor/Fortrans convention of passing scalar variables by address. Each argument to a subroutine must be inspected to see if its value is changed in the subroutine. If it is altered, a temporary variable must be created within the subroutine so that the original variable is not changed. A mechanical translator would probably always create these local variables.

3.3 Software Tools Kermit Program History

Our version of Kermit was implemented using the Software Tools routines. This Kermit was developed on a Sperry (Univac) 1100. It then transferred (by tape) to a Hewlett Packard 3000 by Ken Poulton. It was brought up on the 3000 and enhanced. The Kermit program, naturally enough, was then used to transfer updates between us and Ken Poulton. Ken Poulton also transferred Kermit to Dave Martin. This version of Kermit, when completed, will be the official Software Tools version of Kermit. The features of the Kermit program are not all implemented. Some never will be in a portable way. The following is a description of the features of the Software Tools Kermit and the current state of

development for each feature.

file transfer

The ability to transfer files is the most important feature of Kermit. All other features only help get the job done better or faster. File transfer was thus the first feature implemented.

terminal emulation

Terminal emulation is not currently implemented. Terminal emulation only makes sense on a PC. A similar feature on a main frame is often called "virtual terminal". With the fork primitive, the virtual terminal feature could be written in a few lines of code. Terminal emulation, on the other hand, usually requires much more work. Fortunately, this work has been done for a variety of PCs. If you have a small computer, Kermit may already be written for you and you will not need the Software Tools version of Kermit.

server mode

Some of the server mode commands are implemented. Server mode dramatically simplifies the process of using Kermit. It allows all actions related to file transfer to be accomplished by giving commands to given to the local Kermit program. These commands are translated by the remote Kermit to provide the specified action. The following is a list of server commands.

Command	Action
-----	-----
Send	Send a file
Receive	Receive a file
Login	Sign-on
CD	Change default directory
Logout	Logout
Finish	Stop the Kermit server
Directory	List file names
Disk Usage	Free space
Erase	Delete a file
Type	List the contents of a file
Submit	Submit a job for batch processing
Print	Print a file on a remote printer
Who	Who's logged on
Message	Send a message to someone logged on
Help	Display help information
Query	Remote server status query

The commands send, receive, and finish have been implemented. For the Sperry, the logout command was implemented. Using the Software Tools primitives it should be possible to implement the cd, directory, erase, type, help and query commands.

What is a Domain?

Mark R. Horton

Bell Laboratories
Columbus, Ohio 43213

ABSTRACT

In the past, electronic mail has used many different kinds of syntax, naming a computer and a login name on that computer. A new system, called "domains", is becoming widely used, based on a heirarchical naming scheme. This paper is intended as a quick introduction to domains. For more details, you should read some of the documents referenced at the end.

1. Introduction

What exactly are domains? Basically, they are a way of looking at the world as a heirarchy (tree structure). You're already used to using two tree world models that work pretty well: the telephone system and the post office. Domains form a similar heirarchy for the electronic mail community.

The post office divides the world up geographically, first into countries, then each country divides itself up, those units subdivide, and so on. One such country, the USA, divides into states, which divide into counties (except for certain states, like Louisiana, which divide into things like parishes), the counties subdivide into cities, towns, and townships, which typically divide into streets, the streets divide into lots with addresses, possibly containing room and apartment numbers, the then individual people at that address. So you have an address like

Mark Horton
Room 2C-249
6200 E. Broad St.
Columbus, Ohio, USA

(I'm ignoring the name "AT&T Bell Laboratories" and the zip code, which are redundant information.) Other countries may subdivide differently, for example many small countries do not have states.

The telephone system is similar. Your full phone number might look like 1-614-860-1234 x234. This contains, from left to right, your country code (Surprise! The USA has country code "1"), area code 614 (Central Ohio), 860 (a prefix in the Reynoldsburg C.O.), 1234 (individual phone number), and extension 234. Some phone numbers do not have extensions, but the phone system in the USA has standardized on a 3 digit area code, 3 digit prefix, and 4 digit phone number. Other countries don't use this standard, for example, in the Netherlands a number might be +46 8 7821234 (country code 46, city code 8, number 7821234), in Germany +49 231 7551234, in Sweden +31 80 551234, in Britain +44 227 61234 or +44 506 411234. Note that the country and city codes and telephone numbers are not all the same length, and the punctuation is different from our North American notation. Within a country, the length of the telephone number might depend on the city code. Even within the USA, the length of extensions is not standardized: some places use the last 4 digits of the telephone number for the extension, some use 2 or 3 or 4 digit extensions you must ask an operator for. Each country has established local conventions. But the numbers are unambiguous when dialed from left-to-right, so as long as there is a way to indicate when you are done dialing, there is no problem.

A key difference in philosophy between the two systems is evident from the way addresses and telephone numbers are written. With an address, the most specific information comes

first, the least specific last. (The "root of the tree" is at the right.) With telephones, the least specific information (root) is at the left. The telephone system was designed for machinery that looks at the first few digits, does something with it, and passes the remainder through to the next level. Thus, in effect, you are routing your call through the telephone network. Of course, the exact sequence you dial depends on where you are dialing from - sometimes you must dial 9 or 8 first, to get an international dialtone you must dial 011, if you are calling locally you can (and sometimes must) leave off the 1 and the area code. (This makes life very interesting for people who must design a box to call their home office from any phone in the world.) This type of address is called a "relative address", since the actual address used depends on the location of the sender.

The postal system, on the other hand, allows you to write the same address no matter where the sender is. The address above will get to me from anywhere in the world, even private company mail systems. Yet, some optional abbreviations are possible - I can leave off the USA if I'm mailing within the USA; if I'm in the same city as the address, I can usually just say "city" in place of the last line. This type of address is called an "absolute address", since the unabbreviated form does not depend on the location of the sender.

The ARPANET has evolved with a system of absolute addresses: "user@host" works from any machine. The UUCP network has evolved with a system of relative addresses: "host!user" works from any machine with a direct link to "host", and you have to route your mail through the network to find such a machine. In fact, the "user@host" syntax has become so popular that many sites run mail software that accepts this syntax, looks up "host" in a table, and sends it to the appropriate network for "host". This is a very nice user interface, but it only works well in a small network. Once the set of allowed hosts grows past about 1000 hosts, you run into all sorts of administrative problems.

One problem is that it becomes nearly impossible to keep a table of host names up to date. New machines are being added somewhere in the world every day, and nobody tells you about them. When you try to send mail to a host that isn't in your table (replying to mail you just got from a new host), your mailing software might try to route it to a smarter machine, but without knowing which network to send it to, it can't guess which smarter machine to forward to. Another problem is name space collision - there is nothing to prevent a host on one network from choosing the same name as a host on another network. For example, DEC's ENET has a "vortex" machine, there is also one on UUCP. Both had their names long before the two networks could talk to each other, and neither had to ask the other network for permission to use the name. The problem is compounded when you consider how many computer centers name their machines "A", "B", "C", and so on.

In recognition of this problem, ARPA has established a new way to name computers based on domains. The ARPANET is pioneering the domain convention, and many other computer networks are falling in line, since it is the first naming convention that looks like it really stands a chance of working. The MILNET portion of ARPANET has a domain, CSNET has one, and it appears that Digital, AT&T, and UUCP will be using domains as well. Domains look a lot like postal addresses, with a simple syntax that fits on one line, is easy to type, and is easy for computers to handle. To illustrate, an old routed UUCP address might read "sdscvax!ucbvax!allegro!bosgd!mark". The domain version of this might read "mark@d.osg.cb.att.uucp". The machine is named d.osg.cb.att.uucp (UUCP domain, AT&T company, Columbus site, Operating System Group project, fourth machine.) Of course, this example is somewhat verbose and contrived; it illustrates the hierarchy well, but most people would rather type something like "bosgd.att.uucp" or even "bosgd.uucp", and actual domains are usually set up so that you don't have to type very much.

You may wonder why the single @ sign is present, that is, why the above address does not read "mark.d.osg.cb.att.uucp". In fact, it was originally proposed in this form, and some of the examples in RFC819 do not contain an @ sign. The @ sign is present because some ARPANET

sites felt the strong need for a divider between the domain, which names one or more computers, and the left hand side, which is subject to whatever interpretation the domain chooses. For example, if the ATT domain chooses to address people by full name rather than by their login, an address like "Mark.Horton@ATT.UUCP" makes it clear that some machine in the ATT domain should interpret the string "Mark.Horton", but if the address were "Mark.Horton.ATT.UUCP", routing software might try to find a machine named "Horton" or "Mark.Horton". (By the way, case is ignored in domains, so that "ATT.UUCP" is the same as "att.uucp". To the left of the @ sign, however, a domain can interpret the text any way it wants; case can be ignored or it can be significant.)

It is important to note that **domains are not routes**. Some people look at the number of !'s in the first example and the number of .'s in the second, and assume the latter is being routed from a machine called "uucp" to another called "att" to another called "cb" and so on. While it is possible to set up mail routing software to do this, and indeed in the worst case, even without a reasonable set of tables, this method will always work, the intent is that "d.osg.cb.att.uucp" is the name of a machine, not a path to get there. In particular, domains are absolute addresses, while routes depend on the location of the sender. Some subroutine is charged with figuring out, given a domain based machine name, what to do with it. In a high quality environment like the ARPA Internet, it can query a table or a name server, come up with a 32 bit host number, and connect you directly to that machine. In the UUCP environment, we don't have the concept of two processes on arbitrary machines talking directly, so we forward mail one hop at a time until it gets to the appropriate destination. In this case, the subroutine decides if the name represents the local machine, and if not, decides which of its neighbors to forward the message to.

2. What is a Domain?

So, after all this background, we still haven't said what a domain is. The answer (I hope it's been worth the wait) is that a domain is a subtree of the world tree. For example, "uucp" is a top level domain (that is, a subtree of the "root".) and represents all names and machines beneath it in the tree. "att.uucp" is a subdomain of "uucp", representing all names, machines, and subdomains beneath "att" in the tree. Similarly for "cb.att.uucp", "osg.cb.att.uucp", and even "d.osg.cb.att.uucp" (although "d.osg.cb.att.uucp" is a "leaf" domain, representing only the one machine).

A domain has certain properties. The key property is that it has a "registry". That is, the domain has a list of the names of all immediate subdomains, plus information about how to get to each one. There is also a contact person for the domain. This person is responsible for the domain, keeping the registry up-to-date, serving as a point of contact for outside queries, and setting policy requirements for subdomains. Each subdomain can decide who it will allow to have subdomains, and establish requirements that all subdomains must meet to be included in the registry. For example, the "cb" domain might require all subdomains to be physically located in the AT&T building in Columbus.

ARPA has established certain requirements for top level domains. These requirements specify that there must be a list of all subdomains and contact persons for them, a responsible person who is an authority for the domain (so that if some site does something bad, it can be made to stop), a minimum size (to prevent small domains from being top level), and a pair of nameservers (for redundancy) to provide a directory-assistance facility. Domains can be more lax about the requirements they place on their subdomains, making it harder to be a top level domain than somewhere lower in the tree. Of course, if you are a subdomain, your parent is responsible for you.

One requirement that is NOT present is for unique parents. That is, a machine (or an entire subdomain) need not appear in only one place in the tree. Thus, "cb" might appear both in the "att" domain, and in the "ohio" domain. This allows domains to be structured more flexibly

than just the simple geography used by the postal service and the telephone company; organizations or topography can be used in parallel. (Actually, there are a few instances where this is done in the postal service [overseas military mail] and the telephone system [prefixes can appear in more than one area code, e.g. near Washington D.C., and Silicon Valley].) It also allows domains to split or join up, while remaining upward compatible with their old addresses.

Do all domains represent specific machines? Not necessarily. It's pretty obvious that a full path like "d.cbosg.att.uucp" refers to exactly one machine. The OSG domain might decide that "cbosg.att.uucp" represents a particular gateway machine. Or it might decide that it represents a set of machines, several of which might be gateways. The "att.uucp" domain might decide that several machines, "ihnp4.uucp", "whgwj.uucp", and "hogtw.uucp" are all entry points into "att.uucp". Or it might decide that it just represents a spot in the name space, not a machine. For example, there is no machine corresponding to "arpa" or "uucp", or to the root. Each domain decides for itself. The naming space and the algorithm for getting mail from one machine to another are not closely linked - routing is up to the mail system to figure out, with or without help from the structure of the names.

The domain syntax does allow explicit routes, in case you want to exercise a particular route or some gateway is balking. The syntax is "@dom₁,@dom₂,...,@dom_n:user@domain", for example, @ihnp4.UUCP,@ucbvax.UUCP;:joe@NIC.ARPA, forcing it to be routed through dom₁, dom₂, ..., dom_n, and from dom_n sent to the final address. This behaves exactly like the UUCP ! routing syntax, although it is somewhat more verbose.

By the way, you've no doubt noticed that some forms of electronic addresses read from left-to-right (cbosgd!mark), others read from right-to-left (mark@Berkeley). Which is better? The real answer here is that it's a religious issue, and it doesn't make much difference. left-to-right is probably a bit easier for a computer to deal with because it can understand something on the left and ignore the remainder of the address. (While it's almost as easy for the program to read from right-to-left, the ease of going from left-to-right was probably in the backs of the minds of the designers who invented host:user and host!user.)

On the other hand, I claim that user@host is easier for humans to read, since people tend to start reading from the left and quit as soon as they recognize the login name of the person. Also, a mail program that prints a table of headers may have to truncate the sender's address to make it fit in a fixed number of columns, and it's probably more useful to read "mark@d.osg.a" than "ucbvax!sdcsv".

These are pretty minor issues, after all, humans can adapt to skip to the end of an address, and programs can truncate on the left. But the real problem is that if the world contains BOTH left-to-right and right-to-left syntax, you have ambiguous addresses like xly@z to consider. This single problem turns out to be a killer, and is the best single reason to try to stamp out one in favor of the other.

3. So why are we doing this, anyway?

The current world is full of lots of interesting kinds of mail syntax. The old ARPA "user@host" is still used on the ARPANET by many systems. Explicit routing can sometimes be done with an address like "user@host2@host1" which sends the mail to host1 and lets host1 interpret "user@host2". Addresses with more than one @ were made illegal a few years ago, but many ARPANET hosts depended on them, and the syntax is still being used. UUCP uses "h1h2!h3user", requiring the user to route the mail. Berknets use "host:user" and do not allow explicit routing.

To get mail from one host to another, it had to be routed through gateways. Thus, the address "csvax:mark@Berkeley" from the ARPANET would send the mail to Berkeley, which would forward it to the Berknet address csvax:mark. To send mail to the ARPANET from UUCP, an

address such as "ihnp4!ucbvax!sam@foo-unix" would route it through ihnp4 to ucbvax, which would interpret "sam@foo-unix" as an ARPANET address and pass it along. When the Berknet-UUCP gateway and Berknet-ARPANET gateway were on different machines, addresses such as "csvax:ihnp4!ihnss!warren@Berkeley" were common.

As you can see, the combination of left-to-right UUCP syntax and right-to-left ARPANET syntax makes things pretty complex. Berknets are gone now, but there are lots of gateways between UUCP and the ARPANET and ARPANET-like mail networks. Sending mail to an address for which you only know a path from the ARPANET onto UUCP is even harder — suppose the address you have is ihnp4!ihnss!warren@Berkeley, and you are on host rlgvax which uses seismo as an ARPANET gateway. You must send to seismo!ihnp4!ihnss!warren@Berkeley, which is not only pretty hard to read, but when the recipient tries to reply, it will have no idea where the break in the address between the two UUCP pieces occurs. An ARPANET site routing across the UUCP world to somebody's Ethernet using domains locally will have to send an address something like "xxx@Berkeley.ARPA" to get it to UUCP, then "ihnp4!decvax!island!yyy" to get it to the other ethernet, then "sam@csvax.ISLAND" to get it across their ethernet. The single address would therefore be ihnp4!decvax!island!sam@csvax.ISLAND@Berkeley.ARPA, which is too much to ask any person or mailer to understand. It's even worse: gateways have to deal with ambiguous names like ihnp4!mark@Berkeley, which can be parsed either "(ihnp4!mark)@Berkeley" in accordance with the ARPANET conventions, or "ihnp4!(mark@Berkeley)" as the old UUCP would.

Another very important reason for using domains is that your mailing address becomes absolute instead of relative. It becomes possible to put your electronic address on your business card or in your signature file without worrying about writing six different forms and fifteen hosts that know how to get to yours. It drastically simplifies the job of the reply command in your mail program, and automatic reply code in the netnews software.

4. Further Information

For further information, some of the basic ARPANET reference documents are in order. These can often be found posted to Usenet, or available nearby. They are all available on the ARPANET on host NIC via FTP with login ANONYMOUS, if you have an ARPANET login. They can also be ordered from the Network Information Center, SRI International, Menlo Park, California, 94025.

- RFC819 The Domain Naming Convention for Internet User Applications
- RFC821 Simple Mail Transfer Protocol
- RFC822 Standard for the Format of ARPANET Text Messages
- RFC881 The Domain Names Plan and Schedule

Proposal for a UUCP/Usenet Registry Host

*Mark R. Horton
Karen Summers-Horton
Berry Kercheval*

The following proposal was approved and funded by the Usenix Board in early April. This announcement of the project will be followed by several progress reports on the mapping effort, software development, etc.

A service is proposed to keep up-to-date UUCP and Usenet maps available to the UNIX community, and to make UUCP and Usenet software available in the public domain. The availability of this service will enable domain based electronic mail and also have several other benefits.

With the help of several volunteers, we will create, maintain, and distribute a high quality UUCP map, using data contributed by the UUCP community. This map will be similar to the current Usenet (news) map created and maintained by Karen, but with a format more suited to the UUCP network. (UUCP sites tend to have dozens or even hundreds of UUCP connections, with different costs attached to each connection, while Usenet sites tend to have only a few news connections.) The data will initially be kept in the same format as Steve Bellovin's pathalias program accepts.

The UUCP network changes constantly, and a constant effort to keep this map up-to-date will be needed for the information to be useful.

Karen has been maintaining and distributing the Usenet (news) map for over a year. This effort has been relatively small, compared to the UUCP map, because there are fewer sites on Usenet, and because the connection factor is smaller for Usenet.

Mark will put a copy of the latest Berkeley B News code on the new machine, and recruit several volunteers to help with future maintenance on that machine. This will enable the software to remain in the public domain, since no Bell Labs time, information, or resources will be used for it. This is a small time commitment, since the software currently exists and works, but needs periodic maintenance to fix bugs, and enhancement to deal with the changing nature of Usenet.

In addition, several pieces of public domain software to use UUCP map information to route mail according to the ARPA standards have been posted to Usenet, but are not part of any complete distribution. Several additional routing programs must be written and assembled into a complete package. The resulting package would be put into the public domain and posted to Usenet.

The benefits from this service would be many. The problems described above would be mitigated, if not completely solved.

The B News software will not fall into a state of disrepair. Measures necessary to keep the net from smothering itself with traffic will be possible.

A Reliable Mail Service for the UUCP Net
Implementation Status Report

Berry Kercheval
Zehntel Inc.
2625 Shadelands Drive
Walnut Creek, CA 94598
(415)932-6900

ABSTRACT

An effort is currently underway to implement a reliable mail service for the uucp net. At the writing of this abstract little concrete coding has been done, but there should be substantial progress to report by the date of the meeting.

Goals of the effort are reliable delivery of mail; automatic maintenance of UUCP routing information, with automatic mail routing a natural consequence, and eventual qualification as an ARPA 'domain'.

Current plans call for the project to be implemented in three phases. Phase one will address mail delivery only, understand RFC822 domains at the user level, use the existing UUCP transport mechanism and be upward compatible with UUCP.

Phase two will include an enhanced user interface, automatic route database management and distribution, and document the interchange standards.

Phase three is the pie in the sky phase; no concrete plans currently exist.

What is the Future for 4.2BSD?

Kirk Mckusick, U.C. Berkeley, moderator

Panelists

Robert Elz University of Melbourne

Bill Joy Sun Microsystems

Mike Karels U.C. Berkeley

Sam Leffler Lucasfilms

Bill Shannon Sun Microsystems

Probably others

A distinguished panel of 4.2BSD architects and builders will give their views on the future of 4.2BSD. Following a brief statement by each panelist they will entertain deep philosophical questions from the floor. Due to the stature of the panel, they will *not* answer questions of the form "when will I get my tape?".

Able Computer

Able Computer designs and manufactures an extensive line of communications, general purpose and network connectivity products which are hardware compatible and software transparent to host computer systems manufactured by Digital Equipment Corporation. Able's communications products include multiplexers for PDP-11 and VAX computers: this product group includes the VMZ/32N, a 16 channel multiplexer which emulates the asynchronous line functions of two DEC DMF/32 controllers; the DH/DM, a Unibus controller that provides 16 asynchronous communication lines with modem control; and the DV/16, a 16 line programmable communications multiplexer with modem control.

Able Computer's newest communications product is the ATTACH. ATTACH connects multiple communication lines to one or more DEC PDP or VAX computers. ATTACH supports up to 128 terminals with one hex-size interface. This represents significant savings in power, mounting space and bus loading when compared to equivalent 8 or 16 line multiplexers.

Able Computer also markets memory expansion subsystems such as the ENABLE, designed to allow the expansion of the address space of PDP-11 to 4 megabytes. Bus products include the QNIVERTER, a bi-directional Unibus. Q/Bus signal converter that allows a PDP-11 computer to use LSI peripherals, or vice versa.

Marketing Contact: Melissa Cunliffe
 Able Computer
 1732 Reynolds Avenue
 Irvine, CA 92714
 (714) 979-7030

Alcyon Corporation

Alcyon is a leading manufacturer of LSI-11 bus compatible single board computers (A68KPM) based on the MC68000 microprocessor. The A68KPM provides processing power which is comparable to the LSI 11-73 and exceeds that of the MICRO-VAX. In addition, Alcyon offers various system configurations based on the A68KPM: The APS contained in VT-100-Terminal, the compact Rack Mount System and 30" high cabinet processor. All systems use REGULUS, Alcyon's UNIX compatible operating system.

Marketing Contact: Elaine Bondos
 Alcyon Corporation
 8716 Production Avenue
 San Diego, CA 92121
 (619) 578-0860

Callan Data Systems

Callan will be showing their new Unistar 300, a 68010 based super microcomputer with UNIX System V. Also shown will be the Peripheral Expansion Module providing 474 MBytes of SMD disk storage and industry standard 1/2 inch 9-track tape. Software demonstrated will include Quadratron's Q-One Word Processor, the Unify Relational Database, the DOD certified Telesoft Ada compiler and Sofgram, a Telex, TWX package.

Marketing Contact: Robert Hufnagel
Callan Data Systems
2645 Townsgate Road
Westlake Village, CA 91361
(805) 497-6837

CCA Uniworks

CCA Emacs: Text and programming editor includes multiple windows, multiple editing buffers, automatic compilation of programs and location of errors, Unix shell or VMS PCL in a buffer, and Elisp-extension language utilizing a subset of common lisp.

Marketing Contact: Gwendolyn Whittaker
CCA Uniworks
4 Cambridge Center
Cambridge, MA 02142
(617) 492-8860

Codata Systems Corp.

The CODATA 3300 Series is a powerful 16-bit, 68000-based, MULTIBUS system that can effectively accommodate ten or more users. It's a complete UNIX system that runs full ANSI standard FORTRAN-77, RM/COBOL, BASIC+, SMCBASIC, APL, and PASCAL. The 3300 provides a choice of 12, 33, 84, 168 and 320 Mbytedisk drive configurations and a quad-density 5 1/4" floppy disk system. The system features 320K RAM expandable to 1.5 megabytes. Codata also offers a suite of UNIX based application software including UNIFY and MICRO INGRES.

Marketing Contact: Jonnette Barta, Marketing Communications Coord.
Codata Systems Corp.
285 N. Wolfe Road
Sunnyvale, CA 94086
(408) 735-1744

Computer Technology Group

The Computer Technology Group specializes in UNIX & 'C' training given in public and on-site seminars, as well as offering Video Based and Interactive Videodisc training courses.

CTG's video based training integrates professionally developed and produced video and text materials, including hands-on exercises. CTG courses are produced with the highest standards of video quality, applying the latest techniques of instructional design including the use of computer graphics and animation, compressing learning time.

CTG's interactive videodisc system, co-developed with Interactive Training Systems, Inc. contains computer generated text and graphics designed to direct students and monitor their progress through CTG's video training materials.

Marketing Contact: Janice Duffey
Computer Technology Group
310 South Michigan Avenue
Chicago, Illinois 60604
(312) 987-4084

Data Language Corporation

Data Language Corporation will exhibit the PROGRESS Database Management System and Application Development Language. PROGRESS combines the first "fourth-generation" language available for Unix with a high-performance relational database manager. PROGRESS enables rapid development of large-scale, multi-user applications as well as handling end-user query and reporting needs.

Marketing Contact: Stephen Debler or Joseph Alsop
Data Language Corporation
5 Andover Road
Billerica, MA 01821
(617) 663-5000

Fulcrum Technologies, Inc.

Fulcrum specializes in text management software technology and products. Based on extensive experience with very large systems, Fulcrum has developed a comprehensive full-text information retrieval capability for mini- and micro-computers.

Fulcrum's products are designed for use by managers, researchers, and other such knowledge workers in the office environment. With a simple full-screen interface, this software permits users to access information in a collection of documents of any size merely by specifying combinations of words or phrases that appear anywhere in the documents. Particular attention has been paid to combining broad functionality with very fast retrieval.

This technology is of particular interest to office automation systems integrators, and to those developing applications for vertical markets where the ability to manage unstructured text can provide enhanced functionality or product uniqueness. All software is written in C, and has been implemented on a variety of UNIX-based hardware.

Marketing Contact: Peter Eddison, Director, Marketing
Fulcrum Technologies, Inc.
331 Cooper Street
Ottawa, Ontario, Canada K2P 0G5
(613) 238-1761

Gould Inc.

Gould is exhibiting the power series family of computers - the widest range of Unix-based systems in the world. Demonstrations include the newly announced "FIREBREATHING", the power node 9000 with performance 4.5 times the Vax 11/780 and the powerstation 2000, the first low-cost multi-user Unix system. Gould's "COMPATIBILITY SUITE", application software packages are consistent across the entire power series line.

Marketing Contact: Catherine Baldwin
Gould Inc.
Computer Systems Division
6901 W. Sunrise Boulevard
Ft. Lauderdale, Florida 33313
(305) 587-2900

Human Computing Resources Corp.

Human Computing resources (HCR) is a software and systems house specializing in UNIX software products, applications, systems consulting and interactive computer graphics.

UNITY is HCR's fully licensed implementation of the UNIX system for VAX, PDP-11, Motorola 68000 and National Semiconductor processors.

Chronicle Distribution Management is fully integrated with Chronicle Accounting, and adds Inventory, Invoicing, Sales and Profitability Analysis, Sales Order Processing and Purchase Order Processing to the existing General Ledger, Accounts Receivable and Accounts Payable packages.

The C Compiler Test Suite is a collection of thousands of programs that exhaustively test C constructs. The programs are arranged in modules so that your programmers can test one area in a few minutes or their entire compiler in a few weeks. It conforms to the draft ANSI C standard.

CoCo is a menu driven interactive software project management tool which provides a method of controlling, assigning and evaluating change requests relating to a software package. It can be used efficiently throughout the software development cycle. CoCo maximizes time utilization practices for all members of a project and provides a

framework for task/activity management.

UNIX is a trademark of Bell Labs PDP and VAX are trademarks of Digital Equipment Corporation UNITY, Chronicle and CoCo are trademarks of HCR

Marketing Contact: Jim Peters, Rich McKay
Human Computing Resources Corp.
10 St. Mary Street
Toronto, Ontario M4Y 1P9
Canada
(416) 922-1937

IBC Distribution

IBC/Integrated Business Computers will be demonstrating the High Performance 8 Bit Cadet Multi-User systems and the 16 Bit 32 User Ensign Computer systems. Also demonstrated will be Operating systems software, Oasis, Unix, Application generators, Database Management systems, Graphics and Spreadsheet systems.

Marketing Contact: Garn Tollestrup
IBC Distribution
1140 36th Street, Suite 212
Ogden, Utah 84402
(801) 621-2294/2295

Integrated Solutions, Inc.

High performance 68000/68010 Systems (Optimum Models 5/00 and 5/10) running either Unix 4.2 BSD and/or System III. These systems support local area networking, fast floating point, data base, decision support systems and other software packages. Also shown are sets of 68000/68010 boards. One set is LSI 11 bus compatible and supports Unix 4.2 BSD and/or System III. The second set of boards is VME bus compatible and supports Unix 4.2 BSD.

All boards and systems are based on dual bus architecture with no wait state memory.

In porting Unix 4.2 BSD to Integrated Solutions products one of our major goals was to maintain strict compatibility with the (VAX) 4.2 BSD as distributed by U. C. Berkeley. Evidence of our success can be seen by "trying" our system. Experienced system programmers will find it difficult to tell whether they are logged onto an Integrated Solutions System or a VAX.

Marketing Contact: Stan Coffee
Integrated Solutions, Inc.
2240 Lundy Avenue
San Jose, CA 95131
(408) 943-1902

Intel Corporation

The Intel booth features two XENIX based systems. The System 286/310 is multiuser microcomputer systems designed for the OEM, is based on the high performance iAPX 286 and the industry standard Multibus architecture. Intel will be demonstrating several software packages developed through the Intel independent software vendor programs: Schmidt Kenus, Quadratron Word Processing, SMC Basic, and Microsoft's Multiplan.

The second system is the iDIS, Intel's Speech-Based Information System. It is designed to solve manufacturing quality control problems through word/phrase input to increase data accuracy. The on-line relational database allows for quality control analysis in real time. iDIS is a microcomputer system providing personal computer users the ability to access and share live production data among other PC users and a mainframe computer.

Marketing Contact: Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051
(408) 987-8080

Interlan, Inc.

Interlan, Inc. will show their new single board, NI3210 Multibus Ethernet Controller, featuring an advanced, VLSI-based architecture, with a wide range of UNIX networking software. The board features a unique combination of dual port RAM and an onboard DMA controller, ensuring the highest possible Ethernet network performance while minimizing the service load placed on the host Multibus system.

Available for the NI3210 is Interlan's new XNS Internet Transport Protocols (ITP) for UNIX System V, which provide reliable, flow-controlled communications between UNIX, VAX/VMS, and PDP/RSX-11 systems. The new NI3210 Multibus Ethernet Controller is also supported by UNIX networking packages from UniSoft Systems and Network Research Corporation.

Interlan's NTS10 Terminal Server, a four or eight port unit that interfaces any asynchronous RS-232 device to Ethernet will also be demonstrated.

All of these products are part of Interlan's NET/PLUS product line, providing compatible, reliable, communications and information sharing in a multi-vendor environment.

Marketing Contact: Gerald Wesel
Interlan, Inc.
3 Lyberty Way
Westford, MA 01886
(617) 692-3900

Masscomp

Masscomp will exhibit its MC-500 system and its WorkStation-500. Both use a Masscomp developed Real-Time, Virtual Memory, UNIX-based operating system. Both products are based on a 32-bit VLSI CPU with a 4 KB cache and up to 6 Mb of ECC memory. Independent Graphics Processors may be added to either system to provide very high-performance monochrome or color graphics.

The Masscomp MC-500 computer system architecture incorporates multiple high-speed processors to provide very high system performance. By using multiple processors the system can perform multiple-tasks (e.g., high-speed data acquisition, analysis, graphics, plus software development) without one task degrading another's performance.

The Data Acquisition front-end incorporates bit-slice technology for a 2 megabytes throughput (1 megabyte analog acquisition).

Masscomp has created a system architecture which is able to continually incorporate the most advanced processors, and yet completely protect a user's investment in peripherals and software. Masscomp chose the industry standard UNIX operating system as the base for system software. The system's high-level languages also conform to industry standards. The MULTIBUS IEEE-796 is used for system peripherals such as mass-storage devices.

Marketing Contact: Steve Mullen
 Masscomp
 One Technology Park
 Westford, MA 01886
 (617) 692-6200

Nial Systems Limited

Nial Systems will demonstrate Q'Nial (R), a high-level interactive programming system. A powerful software development tool, Q'Nial is useful for rapid prototyping, modelling of business or scientific systems, relational data base, artificial intelligence or expert systems design. It is available on VAX- UNIX or VMS, SUN, APOLLO, CADMUS, PYRAMID and IBM pc. Q'Nial will be demonstrated on the CADMUS 9000. Franz Inc. is sharing the booth and will demonstrate Franz Lisp on the Sun Workstation. Franz Inc. ports Franz Lisp to Unix workstations, and provides consulting service on Franz Lisp.

'Nial is a registered trademark of Queen's University at Kingston, Ontario.

Marketing Contact: Bill Jenkins
 NIAL Systems Limited
 20 Hatter Street
 Kingston, Ontario, Canada K7M 2L5
 (613) 549-1432

Officesmiths Inc.

The Officesmith - A UNIX-based document management system designed to provide organizations with a solution to the maintenance of information. The Officesmith operates in a structured environment which includes multiple windows, menus, single function key commands and a user programmable help facility. It supports a complete range of document management activities from definition and creation of documents to updating, filing and retrieving information. Combining powerful yet simple application development tools and document management facilities, The Officesmith offers systems integrators and office systems developers the resources to build and maintain responsive corporate information structures.

OfficePolicy - A methodology for documenting, communicating, and accessing policies and procedures in large organizations. OfficePolicy is a complete approach combining management guides, training courses, and software tools based on The Officesmith. The system features automatic indexing/cross referencing, search and retrieval facilities, and consistency in standards and writing format.

Marketing Contact: Glenn A. McInnes, President
Officesmiths Inc.
331 Cooper Street
Ottawa, Ontario
K2P 0G5 CANADA
(613) 235-6749

Pacific Microcomputers, Inc.

Pacific Microcomputers' display will consist of several levels of 68000 single-board microprocessors displaying the latest technology in controlling multi-processor environments, high speed data transfer, no-wait-state processing for OEM applications. In addition, Pacific will exhibit fully-integrated microprocessors with UNIX operating system and full support of all standard programming languages. These systems are highly flexible designs utilizing both 8" and 5 1/4" technology providing high performance aimed at the multi-user marketplace for OEMs and sophisticated end users.

Marketing Contact: Sherrell Harper
Pacific Microcomputers, Inc.
119 Aberdeen Drive
Cardiff, CA 92007
(619) 436-8649

Relational Database Systems, Inc.

Relational Database Systems, Inc. will feature its line of information management software for Unix and MS-DOS computer systems, including the:

INFORMIX Relational Database Management System

Informix provides a suite of comprehensive tools to ease application development, as well as the utilities required to efficiently create and maintain databases.

C-ISAM Indexed Sequential Access Method for C Programmers

C-ISAM is a B+ Tree based access method used to create, manipulate, and retrieve data from indexed files.

File-it!

File-it!, an INFORMIX-compatible File Manager, allows a computer novice to create, maintain, modify and retrieve vital lists of information easily and efficiently.

Marketing Contact: Robert J. Macdonald
Relational Database Systems, Inc.
2471 East Bayshore Road, Suite 600
Palo Alto, CA 94303
(415) 424-1300

The Santa Cruz Operation, Inc.

The Santa Cruz Operation (SCO) features the XENIX operating system and a full range of UNIX-based systems software, and is the exclusive source of XENIX for the IBM PC and the Apple Lisa. SCO's selection of applications software includes the Multiplan spreadsheet package; Uniplex, a comprehensive, menu-driven word processor; fully GSA certified Level II, Cobol; and the Informix relational database management system. SCO also features the UNIX System Tutorials, an introductory training package for new UNIX system users, and an integrated family of cross-assemblers. For the UNIX-system OEM, SCO provides comprehensive system support and software development services.

Marketing Contact: Lisa Smith
The Santa Cruz Operation, Inc.
500 Chestnut Street
Santa Cruz, CA 95060
(408) 425-7222

Sun Microsystems, Inc.

Sun Microsystems is displaying members of its Sun-2 family of SunStations -- high-performance multi-purpose workstations designed for technical professionals to enhance productivity. Each SunStation offers 68010 virtual memory processor 1-4 MB main memory, high-resolution 19-inch bit mapped display, an optical mouse and Sun UNIX 4.2 bsd enhanced with support for graphics and networking. The SunStations can be networked together via Ethernet to form a distributed computing environment, sharing peripherals like mass storage, printers, etc.

The SunStations currently are used in a wide variety of applications including CAD/CAM/CAE, typesetting and documentation preparation, software development, animation, a range of scientific and research applications and artificial intelligence.

Marketing Contact: Wendy Schmidt
 Sun Microsystems, Inc.
 2550 Garcia Avenue
 Mountain View, CA 94043
 (415) 960-1300

Unilogic, Ltd.

Unilogic provides source-code software translations from virtually any high-level programming language to the C language for operation under various UNIX and UNIX-like operating systems. Unilogic is able to perform these translations by use of automated in-house technology which accepts as input many high-level languages, (e.g. COBOL, PL/1, etc.) and produces corresponding C code automatically.

Marketing Contact: Scott W. Wilson
 Unilogic, Ltd.
 160 North Craig Street
 Pittsburgh, PA 15213
 (412) 621-2277

UniPress Software, Inc.

UniPress Software, Inc. is a software publishing house primarily for Unix based environments, as well as support for VMS and MS-DOS environments. We provide programming productivity aids, word processing, spreadsheet and database software. Some of our products include:

Emacs - The acclaimed Gosling version which features full screen editing, multiple windows allowing several files to be edited simultaneously, and extensibility through the compiled MLISP programming language. Emacs runs on the VAX under Unix and VMS, many 68000 implementations, and is now available for the IBM-PC running MS-DOS.

Lex Word Processor - Powerful interactive full screen, menu driven word processor, with "rulers" to format text, full four function calculator, spelling checker, and mass mailing/database system.

Q-Calc and UniCalc Spreadsheets - Q-Calc is an extraordinarily powerful spreadsheet, model size is 999 by 18,000. Interfaces with the Unix environment. Supports graphics for bar and pie charts, graphics and color. UniCalc is a spreadsheet for any Unix system. Model size is 255 by 64. Many commands and extensive help facilities included.

Lattice C Cross compiler for the 8086 family - Lattice C cross compiler is widely regarded as the finest C compiler for the 8086 family. It produces the fastest and tightest code. Package includes C compiler, host linker and librarian, and 8087 support. Small, medium, compact, and large memory models available. The Lattice C Native compiler is available for the IBM-PC running under MS-DOS.

UniPress will also be showing the Unix system V operating system on the Apple

Lisa computer with a wide range of products available for this configuration.

Marketing Contact: Joyce Bielen
UniPress Software, Inc.
Suite 312
2025 Lincoln Highway
Edison, New Jersey 08817
(201) 985-8000

UniSoft Systems Corporation

UniSoft Systems, the leading supplier of UNIX Operating systems for the Motorola 68000 family of microprocessors, will demonstrate UniPlus+ System V. UniPlus+ includes the full system V UNIX Operating system kernel and Utilities, High Performance, record locking, IEEE floating point, C Compiler, System administrator facilities, VI. Also demonstrated will be UniSoft's B-Net, an implementation of TCP/IP supporting layers 3 through 7 of the ISO model. Available applications software packages running under UniPlus+ includes: ADA, ASM68, SVS Fortran, SVS Pascal, SVS Basic Plus, R/M Cobol, SMC Basic, Viewcomp, Lex68.

Marketing Contact: Robert R. Ackerman, Jr., Director of Marketing
UniSoft Systems Corporation
739 Allston Way
Berkeley, CA 94710
(415) 644-1230

Author Index

Appelbe, Bill	111	Matthews, Manton	133
Bach, Maurice J.	174	McCool, Willie	280
Becker, Richard A.	326	McKusick, M. Kirk	237
Beebe, Nelson	354	McNamara, Jean Marie	214
Boyd, Stowe	141	McLeod, Bubbette	344
Buroff, Steven J.	174	Meine, Bill	352
Butler, Thomas W.	253	Miller, Richard	178
Butterfield, David	62	Mills, Charles C.	287
Cole, Allen	356	Moore, Lee	258
Collins, Peter E.	159	Nielsen, Erik Reeh	193
Comer, Douglas E.	42	Norred, Mike	354
Dick-Lauder, Piers	11	Ohkubo, Kiyoki	151
Dronek, Gene	227	Painter, Mark	23
Elz, Robert	11, 183	Pawlowski, Brian	332
Feldman, Stuart I.	xi	Paxson, Vern	353
Filipski, Alan	332	Peterson, Larry L.	42
Gettys, James	72	Phillips, Alex	98
Graves, Wayne	323	Pike, Rob	173
Groundwater, Neil	353	Popek, Gerald	62
Gurwitz, Robert	52	Powell, Michael L.	119
Gusella, Riccardo	78	Querido, Bob	111
Horton, Mark R.	368, 373	Riggle, David W.	23
Hosler, Jay	346	Rosenqvist, Vilhelm	193
Ivie, Evan L.	270	Ruan, Zuwang	87
Jacobson, Van	323	Ruggiero, Mario D.	100
Johnson, Chris Peer	208	Shah, Bakul	110, 192
Kahrs, Mark	258	Slattery, Terry	280
Kamath, Yogeesh	133	Summers-Horton, Karen	373
Karels, Mike	237	Sventek, Joseph	323
Kennedy, Lisa A.	253	Terry, Douglas B.	23
Kercheval, Berry	373, 374	Tichy, Walter F.	87
Killian, Thomas J.	203	Tilson, Michael	1
Kingston, Douglas P. III	32	Torek, Chris	166
Kivolowitz, Perry S.	297	Vaish, Paresh K.	214
Koenig, Andrew	312	Walsh, Robert	52
Kummerfeld, R.J.	11	Warnock, Robert P. III	110, 192, 224
Lai, Clara S.	208	Wasserman, Anthony I.	287
Lankford, Jeffrey P.	228	Weinberger, Peter J.	86
Lauesen, Soren	193	Weinstein, Lauren	18
Lee, Kok-Weng	100	Weiser, Mark	166
Leffler, Sam	237	Zatti, Stefano	78
Leres, Craig	323	Zhou, Songnian	23
Martin, Dave	352		